

# **Discriminative Subgraph Pattern Mining and Its Applications**

Ning Jin

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill

2012

Approved by,  
Diane Pozefsky  
Jan Prins  
Jack Snoeyink  
Alexander Tropsha  
Wei Wang

## **Abstract**

Ning Jin: Discriminative Subgraph Pattern Mining and Its Applications  
(Under the direction of Wei Wang)

My dissertation concentrates on two problems in mining discriminative subgraphs: how to efficiently identify subgraph patterns that discriminate two sets of graphs and how to improve discrimination power of subgraph patterns by allowing flexibility. To achieve high efficiency, I adapted evolutionary computation to subgraph mining and proposed to learn how to prune search space from search history. To allow flexibility, I proposed to loosely assemble small rigid graphs for structural flexibility and I proposed a label relaxation technique for label flexibility.

I evaluated how applications of discriminative subgraphs can benefit from more efficient and effective mining algorithms. Experimental results showed that the proposed algorithms outperform other algorithms in terms of speed. In addition, using discriminative subgraph patterns found by the proposed algorithms leads to competitive or higher classification accuracy than other methods. Allowing structural flexibility enables users to identify subgraph patterns with even higher discrimination power.

## Table of Contents

List of Figures.....	vii
List of Tables.....	x
1 Introduction .....	1
1.1 Motivations.....	2
1.1.1 Protein Active Site Identification and Function Prediction.....	2
1.1.2 Chemical Compound Activity Prediction .....	4
1.2 Related Work on Mining Discriminative Subgraphs .....	4
1.3 Contributions .....	6
1.3.1 Thesis Statement.....	6
1.3.2 Contributions .....	6
1.4 Organization of the Dissertation.....	9
2 Preliminaries.....	10
2.1 Definitions .....	10
2.2 Experimental Setup .....	14
2.3 Datasets for Evaluation.....	15
3 Review of Related Work on Discriminative Subgraph Mining .....	18
3.1 LEAP .....	19
3.1.1 Structural Leap Search .....	19
3.1.2 Frequency Descending Mining.....	20
3.1.3 Overall Framework.....	20

3.2	GraphSig.....	21
3.2.1	Feature Vector Generation .....	21
3.2.2	Significant Sub-feature Vectors .....	21
3.2.3	Overall Framework.....	22
3.3	CORK.....	22
3.3.1	A Submodular Discrimination Score Function .....	22
3.3.2	Overall Framework.....	23
4	Mining Discriminative Subgraph Patterns Using Evolutionary Computation .....	25
4.1	Encoding Subgraphs with Conditional Canonical Adjacency Matrices.....	26
4.2	Mining Discriminative Subgraph Patterns Using Evolutionary Computation .....	29
4.2.1	Framework of the Pattern Evolution: Organization and Resources .....	29
4.2.2	Pattern Extension.....	32
4.2.3	Pattern Migration and Competition .....	33
4.2.4	Generating Consensus Results .....	36
4.3	Experiments .....	37
4.3.1	GAIA Performance Analysis.....	37
4.3.2	Comparison with Other Methods .....	42
5	Mining Discriminative Subgraph Patterns by Learning from Search History .....	49
5.1	Pattern Exploration Order.....	51
5.2	Fast Probing Subgraph Pattern Space.....	56
5.3	Upper-bound Estimation by Learning from Search History .....	58
5.4	Experiments .....	64
5.4.1	Power of Multiple-lineage Exploration .....	64

5.4.2	Efficiency of <i>Fast-probe</i> .....	66
5.4.3	Effectiveness of Using Search History .....	67
5.4.4	Quality of Individual Patterns .....	71
6	Mining Discriminative GG-subgraph Patterns .....	75
6.1	Mining Discriminative GG-subgraph Patterns .....	77
6.1.1	Definition of GG-subgraph Patterns .....	77
6.1.2	Overview of Mining Discriminative GG-subgraph Patterns .....	80
6.1.3	How to Choose Rigid Subgraph Patterns for Transformation .....	81
6.1.4	Phase 1: Discriminative Subgraph Mining and Graph Transformation .....	86
6.1.5	Phase 2: Mining Transformed Graphs for GG-subgraph Patterns .....	88
6.2	Experiments .....	88
6.2.1	Performance Analysis .....	89
6.2.2	Pattern Quality and Runtime Efficiency Comparison .....	91
6.2.3	Application in Graph Classification .....	95
7	Mining Discriminative Subgraph Patterns in Graphs with Continuous Label Values .....	99
7.1	Edge Relaxation .....	100
7.2	Experiments .....	103
7.2.1	Quality of Relaxed Subgraph Patterns .....	103
7.2.2	Classification Accuracy .....	105
7.2.3	Case Study 1: Active Site Identification .....	106
7.2.4	Case Study 2: Function Inference .....	108
8	Conclusions and Future Directions .....	110
8.1	Conclusions .....	110

8.2 Future Directions .....	111
References .....	113

## List of Figures

Figure 2.1: An example of two graphs and a subgraph pattern.....	11
Figure 4.1: An example for graph encoding.....	26
Figure 4.2: Three adjacency matrices of subgraph C-C-N.....	26
Figure 4.3: An illustration of candidate pattern organization.....	31
Figure 4.4: average normalized accuracy vs. $c$ (chemical datasets).....	41
Figure 4.5: average normalized accuracy vs. $c$ (protein datasets).....	41
Figure 4.6: average runtime vs. $c$ (chemical datasets).....	41
Figure 4.7: average runtime vs. $c$ (protein datasets).....	41
Figure 4.8: average runtime vs. number of positive graphs (chemical datasets, number of negative graphs=1600).....	42
Figure 4.9: average runtime vs. number of negative graphs (chemical datasets, number of positive graphs=400).....	42
Figure 4.10: normalized accuracy comparison for balanced chemical datasets between GAIA, COM and graphSig.....	44
Figure 4.11: Runtime comparison for balanced chemical datasets between GAIA, COM and graphSig .....	44
Figure 4.12: Normalized accuracy comparison for balanced protein datasets between GAIA, COM and graphSig.....	45
Figure 4.13: Runtime comparison for balanced protein datasets between GAIA, COM and graphSig .....	45
Figure 4.14: Normalized accuracy comparison for unbalanced chemical datasets between GAIA and COM.....	46
Figure 4.15: Normalized accuracy comparison for protein datasets between GAIA and COM.....	46
Figure 4.16: runtime comparison for unbalanced chemical datasets between GAIA and COM.....	47
Figure 4.17: runtime comparison for unbalanced protein datasets between GAIA and COM .....	47

Figure 5.1: An example of input positive set and negative set.....	52
Figure 5.2: An example of multiple-lineage exploration and single-lineage exploration for graphs in Figure 6.1 .....	53
Figure 5.3: An example of search records and the corresponding <i>prediction tree</i> and <i>prediction table</i> .....	62
Figure 5.4: Normalized accuracy comparison between multiple-lineage exploration-based <i>fast-probe</i> and single-lineage-exploration-based <i>fast-probe</i> using chemical datasets .....	65
Figure 5.5: Runtime comparison between multiple-lineage-exploration-based <i>fast-probe</i> and single-lineage-exploration-based <i>fast-probe</i> using chemical datasets .....	65
Figure 5.6: Normalized accuracy comparison between <i>fast-probe</i> , GAIA, COM and graphSig using chemical datasets .....	67
Figure 5.7: Runtime comparison between <i>fast-probe</i> , GAIA, COM and graphSig using chemical datasets .....	67
Figure 5.8: Normalized accuracy comparison between <i>fast-probe</i> alone and LTS using chemical datasets .....	68
Figure 5.9: Normalized accuracy comparison between <i>fast-probe</i> alone and LTS using protein datasets .....	69
Figure 5.10: Normalized accuracy comparison between LTS, GAIA and COM using protein datasets.....	70
Figure 5.11: Runtime comparison between LTS, GAIA and COM using protein datasets ....	71
Figure 5.12: Optimal score comparison between LTS and LEAP using protein datasets.....	73
Figure 5.13: Runtime comparison between LTS and LEAP using protein datasets .....	73
Figure 5.14: Optimal score comparison between LTS, LEAP and <i>fast-probe</i> using chemical compound datasets .....	74
Figure 5.15: Runtime comparison between LTS and LEAP using chemical compound datasets .....	74
Figure 6.1: An example of GG Transformation .....	79
Figure 6.2: Relationship between positive frequency and average discrimination score of super-patterns.....	83
Figure 6.3: Relationship between discrimination score and discrimination score of super-patterns.....	83



Figure 6.4: Distribution of $R(p_i, p_{i-1})$ .....	85
Figure 6.5: Average best score vs. $k$ (chemical datasets) .....	89
Figure 6.6: Average runtime vs. $k$ (chemical datasets) .....	90
Figure 6.7: Average number of GG-nodes in each GG graph vs. $k$ (chemical datasets).....	90
Figure 6.8: Runtime of phase-1 and phase-2 mining (chemical datasets).....	91
Figure 6.9: Best g-test score comparison (chemical datasets).....	92
Figure 6.10: Runtime comparison (chemical datasets) .....	93
Figure 6.11: A pattern (figure a) in the GG graphs and its various appearances (figures b and c) in the original graphs .....	94
Figure 6.12: Best g-test score comparison (protein datasets).....	95
Figure 6.13: Normalized accuracy comparison (chemical datasets) .....	96
Figure 6.14: Runtime comparison (chemical datasets) .....	97
Figure 6.15: Normalized accuracy comparison (protein datasets) .....	98
Figure 7.1: An example to illustrate edge relaxation .....	103
Figure 7.2: Best g-test score comparison .....	105
Figure 7.3: Normalized accuracy comparison.....	106
Figure 7.4: Positions of the active site and the best pattern found by MSG in 1su1A.....	107

## List of Tables

Table 2.1: List of selected SCOP families.....	16
Table 2.2: List of selected PubChem bioassays .....	17
Table 4.1: Normalized accuracy and average runtime of single-GAIA with different values of $s$ , where $n = 4$ (chemical datasets) .....	38
Table 4.2: Normalized accuracy and average runtime of single-GAIA with different values of $n$ , where $s = 10$ (chemical datasets) .....	39
Table 4.3: Summary of parameters .....	43
Table 4.4: summary of comparison between single-GAIA, parallel-GAIA, COM and graphSig.....	48
Table 5.1: Parameter settings for GAIA, COM and graphSig for chemical datasets.....	66
Table 5.2: Parameter settings for GAIA and COM for protein datasets .....	70
Table 6.1: Parameter settings of GAIA and graphSig.....	95
Table 7.1: Top-5 relaxed discriminative subgraph patterns found by MSG .....	108
Table 8.1: Performance summary.....	111

## **Chapter 1**

### **Introduction**

Many scientific applications search for patterns in complex structural information; when this structural information is represented as graphs, a powerful tool is efficiently mining discriminative subgraphs. Here are three examples: the structures of chemical compounds can be stored as graphs, and with the help of discriminative subgraphs, chemists can predict which compounds are potentially toxic [Helma2004]; 3D protein structures can be stored as graphs, and with the help of discriminative subgraphs, pharmacologists can predict which proteins are able to bind certain ligands and which are not [Bandyopadhyay2006]; program flow information can be represented as graphs and with the help of discriminative subgraphs, computer scientists can identify program bugs and predict which program flows are successful and which are not [Cheng2009].

The discrimination power of a subgraph pattern (how discriminative a subgraph pattern is) can be quantitatively measured by user-specified functions (DEFINITION 2.6) and the task of discriminative subgraph mining is to find subgraphs with the highest function values (discrimination score). There are two problems in discriminative subgraph mining. The first problem is that the task of discriminative subgraph mining is computationally intractable due to the high complexity of possible patterns that could be derived to characterize the graphs. Therefore, increasing attention has been devoted to developing faster discriminative subgraph mining algorithms [Ranu2009, Thoma2009, Yan2008]. The second

problem in discriminative subgraph mining is how to allow flexibility in subgraph patterns. Existing discriminative subgraph mining algorithms use rigid subgraph isomorphism to enumerate subgraphs and compute their frequency. Flexibility (mismatch in labels or connectivity) in subgraph patterns is not permitted. As a result, they are unable to find highly discriminative subgraph patterns with varying appearances in the dataset, especially when the dataset has noise or the discriminative substructure patterns are flexible.

This dissertation investigates four different algorithms to improve runtime efficiency of discriminative subgraph mining and/or to allow flexibility in discriminative subgraphs. In addition, it studies the application of discriminative subgraphs in feature substructure identification and graph classification.

## **1.1 Motivations**

Discriminative subgraphs are subgraphs that appear frequently in one set of graphs but infrequently in another set of graphs. Discriminative subgraphs can capture feature substructures that are specific to a chosen set of graphs (feature substructure identification). In addition, they can be used to differentiate one set of graphs from another (graph classification). As a result, discriminative subgraphs have a wide range of applications in structured data. In this dissertation, I focus on two applications listed below.

### **1.1.1 Protein Active Site Identification and Function Prediction**

Proteins are biological macromolecules that perform important functions such as catalyzing chemical reactions and binding ligands. Many protein functions are performed through active sites, substructures of proteins. Active sites are of great interest to scientists in studying mechanisms of protein functions and designing protein structures with desired functions. Traditionally, active sites are identified through expensive and time-consuming

experiments. Therefore, there is a strong need for computational algorithms for protein active site identification [Yao2003, Chen2006, Fei2010, Huan2006]. One algorithm is to utilize discriminative subgraphs found in protein graphs (graph representations of 3D protein structures) [Huan2003, Huan2004].

A 3D protein structure can be represented by an undirected graph. The protein graph of a 3D protein structure may be generated by creating a node for each amino acid and connecting two nearby amino acids with an edge. Nodes can be labeled with amino acid types and edges can be labeled with distances between amino acids.

Given protein graphs and a chosen function, the protein graphs can be grouped into two sets based on whether the corresponding proteins have the function. Discriminative subgraphs are subgraphs that are frequent in protein graphs of proteins with the chosen function but infrequent in other protein graphs. Such subgraphs are very likely to be parts of or nearby active sites because they are specific to proteins with the chosen function.

In active site identification, it is already known whether a protein has a certain function or not. However, most proteins have unknown functions. Therefore, predicting whether a protein has a certain function is another interesting problem in studying proteins [Bandyopadhyay2009]. One solution to protein function prediction is to convert the problem to a graph classification problem [Fei2008, Saigo2008, Fei2009]. In a graph classification problem, the input is two sets of graphs and the output is a computational model that predicts which graph set a graph belongs to. The prediction model is then used to make predictions for graphs that are not present in the input. To convert a protein function prediction problem to a graph classification problem, proteins with known functions are represented by protein graphs and the graphs are grouped into two sets based on whether they have the function.

Then prediction models are generated with discriminative subgraphs being features or even building blocks of the models.

### **1.1.2 Chemical Compound Activity Prediction**

In drug discovery, the search space of candidate chemical compounds is prohibitively large and it is expensive and time-consuming to perform experiments to test activity. Therefore, fast algorithms are strongly needed to predict chemical compound activity and screen candidate compounds [Fröhlich2005, Smalter2008, Smalter2009]. Such algorithms usually calculate prediction models based on a set of selected molecular descriptors that help quantitatively characterize chemical compounds [Ranu2009]. Some of the molecular descriptors can be derived from discriminative subgraphs frequently found in graphs of active compounds but infrequently in graphs of inactive compounds.

A chemical compound structure can be represented by an undirected graph. One way to generate a graph representation for a chemical compound structure is to create a node for each atom and connect two bonded atoms with an edge. Nodes are labeled with atom types and edges are labeled with bond types. Stereo-chemical information may be embedded in node and edge labels if needed.

## **1.2 Related Work on Mining Discriminative Subgraphs**

One straightforward algorithm to find discriminative subgraphs is to first enumerate all the subgraphs that are frequent in one set of graphs and then among the frequent subgraphs select those that are infrequent in the other set of graphs. This exhaustive enumeration and selection approach guarantees to find all discriminative subgraphs. However, the enumeration step typically generates an enormous quantity of candidate subgraphs and computing frequency in the selection step involves subgraph isomorphism,

which is known to be an NP-complete problem. These two limitations prevents this straightforward algorithm from handling large real-world graph datasets. In addition, many subgraphs that are frequent in one set are also frequent in the other set. As a result, the straightforward algorithm is inefficient because much of the computation to enumerate frequent subgraphs is wasted.

To overcome the limitations, three recent algorithms search directly for discriminative subgraph patterns.

LEAP [Yan2008] is a pioneer in discriminative subgraph pattern mining. It looks for the optimal subgraph pattern in terms of discrimination power with a branch-and-bound technique, taking advantage of the fact that structurally similar subgraphs tend to have similar discrimination power. It also uses a technique called “frequency descending mining” to exploit the correlation between subgraph frequency and subgraph discrimination power.

CORK [Thoma2009] proposes to use correspondence to measure the discrimination power of subgraph patterns and thereby achieves a theoretically near-optimal solution. Given a set of subgraph patterns, the number of correspondences is the total number of pairs of graphs that these subgraphs cannot discriminate.

GraphSig [Ranu2009] utilizes frequent subgraph mining to find discriminative subgraphs but in a different way than the straightforward solution. It first converts graphs to feature vectors by performing Random Walk with Restarts on each node. Then it divides graphs into small groups such that graphs in the same group have similar vectors. It mines frequent subgraphs in each group with high frequency thresholds because high similarity in vectors in the same group indicates that the corresponding graphs in the group share highly frequent subgraphs. Using high frequency thresholds in frequent subgraph mining avoids

enumerating a prohibitively large number of candidate subgraphs and enables graphSig to process relatively large datasets efficiently.

All these three algorithms outperform the straightforward algorithm significantly, but they are still not efficient enough when processing large graph datasets. In addition, all of them use rigid subgraph isomorphism and thus are unable to identify discriminative subgraph patterns with varying appearances in graphs.

### **1.3 Contributions**

#### **1.3.1 Thesis Statement**

My proposed discriminative subgraph pattern mining algorithms outperform other state-of-the-art algorithms in terms of:

1. Runtime efficiency
2. Pattern discrimination power measured by discrimination score
3. Classification accuracy

The better performance is demonstrated through experiments with protein and chemical compound structure data.

#### **1.3.2 Contributions**

This dissertation concentrates on three problems in discriminative subgraph mining: (1) how to efficiently identify subgraph patterns that discriminate two sets of graphs, (2) how to improve discrimination power of subgraph patterns by allowing structural flexibility and (3) how applications of discriminative subgraphs can benefit from more efficient and more effective mining algorithms. Below I briefly describe the contributions I made for these problems.



To improve efficiency of discriminative subgraph mining, I proposed two algorithms: GAIA [Jin2010] and LTS [Jin2011].

GAIA employs a novel subgraph encoding algorithm to support an arbitrary subgraph pattern exploration order and explores the subgraph pattern space in a heuristic mining process resembling biological evolution. In this mining process, new candidate patterns are calculated by extending old candidate patterns and candidate patterns with lower discrimination power are more likely to be pruned by the algorithm. In this manner, GAIA is able to find discriminative subgraph patterns much faster than other algorithms. Additionally, it takes advantage of parallel computing to further improve the efficiency of the mining process.

LTS is based on an observation that search history of discriminative subgraph mining is very useful in computing empirical upper bounds of discrimination power of subgraphs. LTS begins with a greedy algorithm that first samples the search space and then calculates a model to estimate upper bounds of discrimination power of subgraphs based on the samples. In the end, LTS explores the search space again in a branch and bound fashion leveraging the upper bound estimation model.

These two algorithms outperform existing algorithms in terms of efficiency. However, they are unable to improve subgraph pattern discrimination power, which is limited by the rigid definition of subgraphs. To overcome this problem, flexibility needs to be allowed in subgraph patterns. There are two types of subgraph flexibility: label flexibility and structural flexibility. To incorporate these two types of flexibility, I proposed two algorithms: MSG and GG-miner.

MSG is an algorithm specifically designed to find flexible discriminative subgraph patterns in protein graphs. Unlike other types of graphs such as chemical compound graphs, protein graphs contain edges labeled with continuous values. Graphs with continuous edge labels have to be discretized first in order to be processed effectively by rigid subgraph mining algorithms. To handle protein graphs with continuous edge labels, MSG searches for subgraph patterns with continuous edge labels instead of discrete labels and associates each subgraph pattern with an RMSD threshold to optimize its discrimination power. The algorithm first enumerates a large number of candidate discriminative patterns with discrete labels by invoking GAIA and then performs edge relaxation on the candidate patterns to generate patterns with continuous edge labels. Experimental results show that patterns found by MSG have higher discrimination power than rigid subgraph patterns using discrete labels.

GG-miner is an algorithm designed to find discriminative subgraph patterns with structural flexibility allowed. To allow structural flexibility, I proposed a new type of substructure patterns: GG-subgraph patterns. In a GG-subgraph pattern, each node corresponds to a rigid subgraph pattern and each edge is labeled with whether the two corresponding subgraph patterns are connected. The lack of detailed descriptions in edge labels allows structural flexibility because edge labels do not describe how the corresponding subgraph patterns are connected. Experimental results show that, due to their structural flexibility, GG-subgraph patterns often have higher discrimination power than the optimal rigid discriminative subgraph patterns.

To evaluate how applications of discriminative subgraphs can benefit from more efficient and effective mining algorithms, I applied the proposed algorithms to solve three real-world problems: protein classification, protein active site identification and chemical

compound activity classification. Experimental results show that the proposed algorithms outperform other algorithms in terms of speed, which enable users to process large databases faster. In addition, using discriminative subgraph patterns found by the proposed algorithms leads to competitive or higher classification accuracy than other algorithms. Besides, allowing label flexibility enables users to identify protein active sites that other algorithms cannot find. Allowing structural flexibility enables users to identify subgraph patterns with higher discrimination power than the optimal patterns that can be found by any rigid subgraph mining algorithm.

#### **1.4 Organization of the Dissertation**

The remainder of this dissertation is organized as follows. Chapter 2 introduces basic graph mining concepts that are fundamental to understanding the remainder of the text. It also explains the experimental setup and the datasets used for evaluation. Chapter 3 reviews related work regarding discriminative subgraph pattern mining. For readers who are familiar with graph mining, these two chapters may be skipped. Chapter 5 and 6 present 2 discriminative subgraph mining algorithms that aim at improving efficiency. Chapter 7 and 8 present 2 algorithms that search for discriminative subgraphs with flexibility allowed. At the end of each chapter that presents an algorithm, the algorithm is applied to at least two applications and its efficiency and effectiveness is evaluated.

## Chapter 2

### Preliminaries

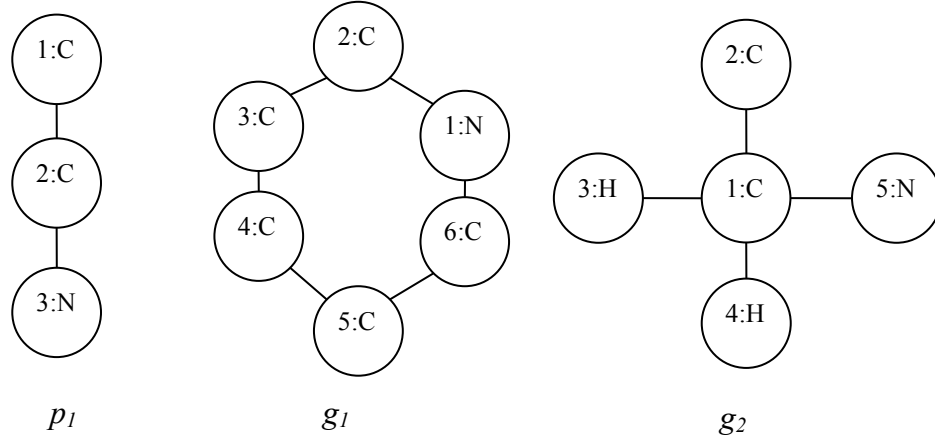
This chapter provides definitions of terms frequently used in this dissertation. It also describes the experimental setup and datasets for evaluation of the proposed algorithms.

#### 2.1 Definitions

**DEFINITION 2.1 (Graph).** A graph is denoted as  $g = (V, E)$  where  $V$  is a set of nodes and  $E$  is a set of edges connecting the nodes. Both nodes and edges can have labels. I use  $g.V$  and  $g.E$  to denote the node set and edge set of  $g$ . I simplify the notation by using only  $V$  and  $E$  to denote the node set and edge set when there is no ambiguity.

Each graph in the graph database has a unique graph ID starting from 1. In a graph, each node has a unique ID starting from 1.

In the example of Figure 2.1, there are two graphs in the graph database with IDs 1 and 2 respectively. Nodes are circles, each containing (node ID : node label). Two nodes in a graph may have the same label but they cannot have the same node ID. Two nodes in two different graphs can have the same node ID but they do not necessarily represent the same entity and may have different labels.



**Figure 2.1: An example of two graphs and a subgraph pattern**

**DEFINITION 2.2 (Subgraph Isomorphism and Graph Isomorphism).** The label of a node  $u$  is denoted by  $l(u)$  and the label of an edge  $(u, v)$  is denoted by  $l((u, v))$ . For two graphs  $g$  and  $g'$ , if there exists an injection  $f: g.V \rightarrow g'.V$  such that for any node  $u \in g.V$ ,  $l(u) = l(f(u))$  and for any edge  $(u, v) \in g.E$ ,  $l((u, v)) = l((f(u), f(v)))$ , then  $g$  is a subgraph of  $g'$ , denoted as  $g \subseteq g'$ , and  $g'$  is a supergraph of  $g$ . I also say  $g'$  supports or contains  $g$ . If  $g$  is a subgraph of  $g'$  and  $g'$  is a subgraph of  $g$ , then  $g$  is isomorphic to  $g'$ .

For example, in Figure 2.1, pattern  $p_l$  is a subgraph of  $g_l$  and of  $g_2$ .

**DEFINITION 2.3 (Embedding).** Given two graphs  $g$  and  $g'$ , an embedding (or occurrence) of  $g$  in  $g'$  is a subgraph  $m = (V', E')$ , where  $V' \subseteq g'.V$ ,  $E' \subseteq g'.E$  and  $m$  is isomorphic to  $g$ .  $g$  may have multiple embeddings in  $g'$ .

**DEFINITION 2.4 (Embedding Code).** Given a graph  $g$  and a subgraph pattern  $p$ , where  $p$  is a subgraph of  $g$ , an embedding code of  $p$  is the concatenation of the graph ID of  $g$

and the sorted list of all node IDs of one of the embeddings of  $p$  in  $g$ . The first element in an embedding is the graph ID and the remaining elements are node IDs.

For example, in Figure 2.1, pattern  $p_1$  has two embeddings in graph  $g_1$ , whose nodes are  $\{1, 2, 3\}$  and  $\{1, 5, 6\}$  respectively, and one embedding in graph  $g_2$ , whose nodes are  $\{1, 2, 5\}$ . In total, pattern  $p_1$  has three embedding codes:  $\langle 1, 1, 2, 3 \rangle$ ,  $\langle 1, 1, 5, 6 \rangle$  and  $\langle 2, 1, 2, 5 \rangle$ .

**DEFINITION 2.5 (Frequency).** Given a graph set  $G$ , the frequency of a subgraph pattern  $p$  is the ratio of the number of graphs supporting  $p$  in  $G$  to the total number of graphs in  $G$ .

The input of a discriminative subgraph pattern mining problem is composed of two sets of graphs: a positive set  $G_p$  and a negative set  $G_n$ .

Given a certain property  $A$ , a positive set is a set of objects with property  $A$  (i.e. having positive test results for the property); the corresponding negative set is a set of objects without property  $A$  (i.e. having negative test results for the property).

For example, in a chemical graph database, a positive set is composed of graphs representing chemical compounds that are active in a given bioassay and the corresponding negative set is composed of graphs representing chemical compounds that are inactive in this bioassay.

I denote the frequency of pattern  $p$  in the positive set by  $pfreq(p)$  and the frequency in the negative set by  $nfreq(p)$ .

**DEFINITION 2.6 (Discrimination Score).** The discrimination score of a subgraph pattern  $p$  is a user-specified function of its frequencies in the positive and negative sets. The

more discriminative the pattern, the larger the discrimination score. The user-specified function should have the following property:

$$\text{when } pfreq(p) > nfreq(p) : \frac{\partial score(p)}{\partial pfreq(p)} > 0, \frac{\partial score(p)}{\partial nfreq(p)} < 0$$

By default, I use the log ratio of positive frequency to negative frequency to measure pattern discrimination power:

$$score(p) = \log \frac{pfreq(p)}{nfreq(p)}$$

To solve the problem of denominator being zero, when calculating the negative frequencies, I added an imaginary negative graph that has all subgraph patterns. Thus, the negative frequency of any pattern  $p$  is never zero.

This function is asymmetric w.r.t. positive and negative frequencies. It concentrates on subgraph patterns with high positive frequency and low negative frequency rather than patterns with low positive frequency and high negative frequency. The rationale is that in many applications, such as structure-based protein and chemical compound classification, positive graphs are much more likely to share common discriminative subgraph patterns than negative graphs. This is because that positive graphs typically have some common characteristics (e.g. a biological function) while negative graphs (e.g., those lacking a biological function) are often highly diverse. However, even if negative graphs share some common discriminative subgraph patterns, these patterns can be found easily by switching the roles of positive graphs and negative graphs.

When I compared the proposed algorithms with LEAP [Yan2008] in terms of discrimination power of resulting patterns, I adopted g-test score as the measurement of

discrimination power for fair comparison because g-test score is the measurement used and optimized in the implementation of LEAP. The definition of g-test score is as follows:

$$\text{g-test score of } p = pfreq(p) * \log \frac{pfreq(p)}{nfreq(p)} + (1 - pfreq(p)) * \log \frac{1 - pfreq(p)}{1 - nfreq(p)}$$

In this dissertation, I consider discriminative subgraph pattern mining as a process to search a positive graph set and negative graph set for the subgraph pattern with the highest discrimination score for each positive graph.

## 2.2 Experimental Setup

All the algorithms were implemented in C++ and compiled with g++ with -O2 optimization. All the experiments were performed on a 2.20 GHz dual-core and 3.7 GB memory PC running Ubuntu Linux 64-bit version.

I evaluated the algorithms by their runtime efficiency and the discrimination power of their resulting subgraph patterns. In addition, I used the discriminative subgraph patterns found by the proposed algorithms to generate graph classifiers and evaluate the algorithms by the corresponding classification accuracy.

To generate graph classifiers based on the discriminative GG-subgraph patterns found by GG-miner, I generated a classification rule for each pattern  $p$  in the form of “if  $g$  contains pattern  $p \rightarrow$  graph  $g$  is positive”. Then I selected classification rules to compose graph classifiers, optimizing the normalized accuracy in the training sets. The normalized accuracy is defined as follows.

$$\text{Sensitivity} = (\text{number of true positives}) / (\text{number of positives})$$

$$\text{Specificity} = (\text{number of true negatives}) / (\text{number of negatives})$$

$$\text{Normalized accuracy} = (\text{sensitivity} + \text{specificity}) / 2$$



The rationale for using normalized accuracy is to prevent the result from being biased by unbalanced datasets.

All classification experiments are performed with 5-fold cross validation and the average normalized accuracy over 5 runs is reported for each dataset.

### **2.3 Datasets for Evaluation**

I used protein datasets and chemical compound datasets in my experiments for evaluation.

The protein datasets consist of protein structures from Protein Data Bank<sup>1</sup> classified by SCOP<sup>2</sup> (Structural Classification of Proteins). As for protein datasets, I selected all large SCOP families with more than 25 members (listed in Table 2.1). In each dataset, protein structures in a selected family are taken as the positive set. Unless otherwise specified, I randomly selected 250 outsider proteins (i.e., not members of the 16 families) as a common negative set used by all 16 protein datasets. To generate a protein graph, each graph node denotes an amino acid, whose location is represented by the location of its alpha carbon. There is an edge between two nodes if the distance between the alpha carbons of two amino acids is less than 11.5 angstroms. Nodes are labeled with their amino acid type and edges are labeled with the discretized distance between the alpha carbons. On average, each protein graph has approximately 250 nodes and 2700 edges.

---

1. <http://www.rcsb.org/pdb>

2. <http://scop.mrc-lmb.cam.ac.uk/scop/>

The chemical compound datasets consist of chemical compound structures from PubChem<sup>3</sup> classified by their biological activities, listed in Table 2.2. Each compound can be either active or inactive in a bioassay. The same datasets were used in [Yan2008] and [Ranu2009]. For each bioassay, I randomly selected 400 active compounds as the positive set and 400 inactive compounds as the negative set. I generated balanced chemical compound datasets in order to compare with graphSig [Ranu2009] whose implementation can only process balanced datasets. Therefore, the balanced chemical datasets are the default chemical datasets in experiments. In chemical compound graphs, each atom is represented by a graph node labeled with the atom type and each chemical bond is represented by a graph edge labeled with the bond type. On average, each compound graph has 54.76 nodes and 57.24 edges.

**Table 2.1: List of selected SCOP families**

SCOP ID	Family name	# of proteins
46463	Globins	51
47617	Glutathione S-transferase (GST)	36
48623	Vertebrate phospholipase A2	29
48942	C1 set domains	38
50514	Eukaryotic proteases	44
51012	alpha-Amylases, C-terminal beta-sheet domain	26
51487	beta-glycanases	32
51751	Tyrosine-dependent oxidoreductases	65
51800	Glyceraldehyde-3-phosphate dehydrogenase-like	34
52541	Nucleotide and nucleoside kinases	27
52592	G proteins	33
53851	Phosphate binding protein-like	32
56251	Proteasome subunits	35
56437	C-type lectin domains	38
88634	Picornaviridae-like VP	39
88854	Protein kinases, catalytic subunit	41

---

3. <http://pubchem.ncbi.nlm.nih.org>

**Table 2.2: List of selected PubChem bioassays**

Bioassay ID	Tumor description	# of actives	# of inactives
1	Non-Small Cell Lung	2047	38410
33	Melanoma	1642	38456
41	Prostate	1568	25967
47	Central Nerv Sys	2018	38350
81	Colon	2401	38236
83	Breast	2287	25510
109	Ovarian	2072	38551
123	Leukemia	3123	36741
145	Renal	1948	38157
167	Yeast anticancer	9467	69998
330	Leukemia	2194	38799

## **Chapter 3**

### **Review of Related Work on Discriminative Subgraph Mining**

This chapter reviews three other discriminative subgraph pattern mining algorithms: LEAP [Yan2008], graphSig [Ranu2009] and CORK [Thoma2009]. Each of the three algorithms tackles the problem of discriminative subgraph pattern mining from a unique perspective.

LEAP uses branch and bound search to mine discriminative subgraph patterns. The authors proposed two techniques to help pruning search space: structural leap search and frequency descending mining. Structural leap search takes advantage of the fact that subgraphs with similar structures have similar discrimination power. Frequency-descending mining is motivated by an observation that subgraphs with higher frequency are more likely to be discriminative than average.

GraphSig uses frequent subgraph mining to mine discriminative subgraph patterns. As mentioned in Chapter 1.2, it is infeasible to first enumerate all frequent subgraphs and then select discriminative ones because of the explosive number of candidate patterns. GraphSig solves this problem by dividing the input graph dataset into smaller groups. It first converts graphs to feature vectors by performing Random Walk with Restarts on each node. Then it divides graphs into small groups such that graphs in the same group have similar vectors. It mines frequent subgraphs in each group with high frequency thresholds because high similarity in vectors in the same group indicates that the corresponding graphs in the

group share highly frequent subgraphs. In the end, it performs feature selection to find highly discriminative subgraph patterns among candidate frequent subgraph patterns.

CORK is different from LEAP and graphSig in that CORK aims at optimizing the discrimination power of a set of subgraph patterns instead of individual subgraph patterns. CORK uses a greedy algorithm to search for a set of discriminative subgraph patterns. However, the algorithm is theoretically guarantees finding near-optimal solutions because CORK chooses a specific scoring function to measure the discrimination power of a set of subgraph patterns. The scoring function chosen by CORK is the number of correspondences. Given a set of subgraph patterns, the number of correspondences is the total number of pairs of graphs that these subgraphs cannot discriminate.

### 3.1 LEAP

#### 3.1.1 Structural Leap Search

Yan et al. [Yan2008] made an observation in branch and bound search for discriminative subgraphs: if two subgraph patterns are highly similar in their structures, then there is usually strong similarity in their positive and negative frequencies as well. As a result, their discrimination scores should also be similar. Therefore, if a subgraph pattern  $p$  has already been explored and subgraph pattern  $q$  is similar to  $p$ , pattern  $q$  can be skipped.

The similarity between two subgraph patterns  $p$  and  $q$  is measured by the ratio of the maximum frequency difference that  $p$  and  $q$  can have to the sum of frequencies of  $p$  and  $q$ . If the ratio is less than a user specified threshold  $\sigma$ , then the two subgraph patterns are considered highly similar and there is no need to explore the other if one is already explored. Let  $\Delta_p(p, q)$  be the maximum positive frequency difference that  $p$  and  $q$  can have and  $\Delta_n(p, q)$

be the maximum negative frequency difference that  $p$  and  $q$  can have. After one pattern is explored, the other can be skipped if:

$$\frac{2\Delta_p(p,q)}{pfreq(p) + pfreq(q)} < \sigma \text{ and } \frac{2\Delta_n(p,q)}{nfreq(p) + nfreq(q)} < \sigma$$

This subgraph pattern pruning can be further extended to prune a whole search branch instead of an individual subgraph pattern.

### 3.1.2 Frequency Descending Mining

Yan et al. [Yan2008] discovered that if all subgraphs are sorted in ascending order of their frequency, discriminative subgraph patterns are often in the high-end range. To profit from this discovery, the authors proposed an iterative frequency descending mining algorithm.

Frequency descending mining begins the mining process with high frequency threshold  $\theta = 1.0$  and it searches for the most discriminative subgraph pattern  $p^*$  whose frequency is at least  $\theta$ . Then frequency descending mining repeatedly lower the frequency threshold  $\theta$  to check whether it can find better  $p^*$  whose frequency is at least  $\theta$ . It terminates when  $\theta$  reaches either 0 or a user-specified threshold.

### 3.1.3 Overall Framework

The overall framework of LEAP is as follows:

Step 1: Use structural leap search to find the most discriminative subgraph pattern  $p^*$  with frequency threshold  $\theta = 1.0$ ,

Step 2: Repeat Step 1 with  $\theta = \theta / 2$  until  $score(p^*)$  converges,

Step 3: Take  $score(p^*)$  as a seed score; use structural leap search to find the most discriminative subgraph pattern without frequency threshold.

## 3.2 GraphSig

### 3.2.1 Feature Vector Generation

GraphSig predefines a set of simple structural features such as nodes and edges with specific labels. It represents each node in each graph with a feature vector based on the predefined features. As a result, a graph with  $n$  nodes is represented with  $n$  feature vectors. The feature vector for a node reflects the distribution of features around the node.

The algorithm begins with generating feature vectors. A feature vector is generated by performing RWR (Random Walk with Restarts) on a node in each graph. RWR simulates the trajectory of a walker that begins from the starting node and moves from one node to a randomly selected neighbor. Each neighbor has the same probability of being selected for next move. In addition, graphSig limits the distance of random walk by having a restart probability to bring the walker back to the starting node. Each feature value is the probability of it being traversed in RWR. Therefore, a high feature value means the feature is close to the starting node. When the feature values converge, RWR terminates.

### 3.2.2 Significant Sub-feature Vectors

Given two feature vectors  $x = \langle x_1, x_2, \dots, x_m \rangle$  and  $y = \langle y_1, y_2, \dots, y_m \rangle$ , graphSig defines that  $x$  is called a sub-feature vector of  $y$  if and only if  $x_i \leq y_i$ , for  $i = 1, \dots, m$ .

The probability of a feature vector  $x$  occurring in a random feature vector  $y$  is calculated as follows:

$$P(x) = \prod_{i=1}^m P(y_i \geq x_i)$$

Once the probability of feature vector  $x$  occurring in a random feature vector is known, the p-value of feature vector  $x$  can be calculated. The smaller the p-value of feature vector  $x$ , the more statistically significant the vector is.

GraphSig searches all feature vectors generated by RWR for common statistically significant sub-feature vectors. A statistically significant sub-feature vector indicates potential existence of discriminative subgraph patterns around the corresponding node. Therefore, for each significant sub-feature vector, graphSig invokes frequent subgraph pattern mining to search the supporting graphs for highly frequent subgraph patterns around the node associated with the sub-feature vector. This frequent subgraph mining process is highly efficient because the number of supporting graphs is very small and the search is limited to the neighborhood around the node associated with the sub-feature vector. In the end, discriminative subgraph patterns can be selected from the frequent subgraph patterns.

### **3.2.3 Overall Framework**

The overall framework of graphSig is as follows:

Step 1: Calculate feature vectors for all graphs with Random Walk with Restarts,

Step 2: Use feature vector mining to find significant and frequent sub-feature vectors with user-specified frequency and p-value thresholds,

Step 3: Use frequent subgraph pattern mining to search graphs that share significant sub-feature vectors for discriminative subgraph patterns.

## **3.3 CORK**

### **3.3.1 A Submodular Discrimination Score Function**

The goal of CORK is to find a subgraph pattern set that can discriminate two sets of graphs instead of individually discriminative subgraph patterns. Therefore, the discrimination



score function used by CORK evaluates the discrimination power of a set of subgraph patterns rather than individual patterns.

CORK uses a greedy algorithm to search for the target pattern set. It begins with an empty pattern set and gradually adds one subgraph pattern to the set at a time. Each time it adds a subgraph pattern to the pattern set, CORK chooses the subgraph pattern that can maximize the discrimination score function of the new pattern set. In general, this greedy algorithm does not guarantee the optimal solution. However, it can guarantee a near-optimal solution if the discrimination score function is submodular. Submodularity is defined as follows:

Given a search space  $D$ , a pattern  $p \in D$ , and two candidate pattern set  $T$  and  $T'$ ,  $T' \subset T \subseteq D$ , a scoring function *score* is submodular if:

$$score(T' \cup \{p\}) - score(T') \geq score(T \cup \{p\}) - score(T)$$

If the scoring function is submodular, it has been proved that the greedy algorithm yields a near-optimal solution and its score achieves at least  $(1 - \frac{1}{e}) \approx 63\%$  of the score of the optimal solution.

Therefore, CORK uses the number of correspondences as its scoring function, which is submodular. Given a set of subgraph patterns, the number of correspondences is the total number of pairs of graphs that these subgraphs cannot discriminate.

### 3.3.2 Overall Framework

The overall framework of CORK is as follows:

Step 1: Initialize the resulting pattern set  $T$  as empty,

Step 2: Select subgraph pattern  $p$  that maximizes  $T \cup \{p\}$ ,

Step 3: If  $score(T \cup \{p\}) > score(T)$ , then insert  $p$  into  $T$  and go to Step 2; otherwise, return the resulting subgraph pattern set  $T$

## Chapter 4

### Mining Discriminative Subgraph Patterns Using Evolutionary Computation

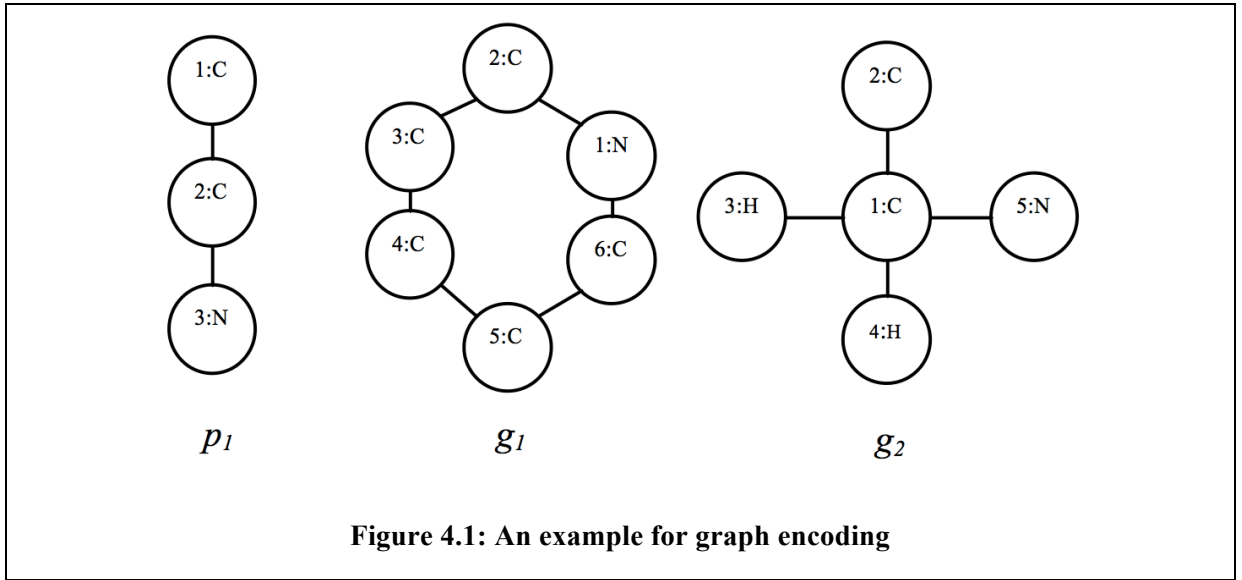
I proposed a novel algorithm GAIA (Graph clAssification with evolutionary computation) [Jin2010] to apply evolutionary computation, which is a randomized searching strategy for optimal solution, to look for discriminative subgraph patterns. Using evolutionary computation enables GAIA to take advantage of the more and more widely available parallel computing resources. The quality of resulting subgraph patterns is improved by running many instances of the algorithm in parallel and then generating a consensus result that has better discrimination power than any resulting set from an individual execution.

The major difficulty of using evolutionary computation to find discriminative subgraphs is that there is no existing subgraph exploration algorithm that can explore subgraph patterns randomly and track such exploration in an efficient way, which is essential to applying evolutionary computation. To overcome this difficulty, I proposed a novel subgraph encoding method using the notion of conditional canonical adjacency matrix. Given a graph database and the embedding information of a subgraph pattern, the proposed encoding method is able to calculate its canonical sequence representation in  $O(|V|^2)$  instead of the exponential time needed in previous methods, where  $|V|$  is the number of nodes in the pattern.

#### 4.1 Encoding Subgraphs with Conditional Canonical Adjacency Matrices

**DEFINITION 5.1 (Conditional Canonical Adjacency Matrix).** Given a graph database, where each graph has a unique graph ID, the conditional canonical adjacency matrix of a subgraph pattern  $p$  is the adjacency matrix corresponding to the lexicographically smallest embedding code of  $p$ .

For example, in Figure 4.1, given the graph database composed of  $g_1$  and  $g_2$ , the conditional canonical adjacency matrix of  $p_1$  is Matrix 1 in Figure 4.2.



<table border="1"> <tr><td>N</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>C</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>C</td></tr> </table> <p>Matrix 1: &lt;1, 1, 2, 3&gt;</p>	N	1	0	1	C	1	0	1	C	<table border="1"> <tr><td>N</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>C</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>C</td></tr> </table> <p>Matrix 2: &lt;1, 1, 5, 6&gt;</p>	N	0	1	0	C	1	1	1	C	<table border="1"> <tr><td>C</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>C</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>N</td></tr> </table> <p>Matrix 3: &lt;2, 1, 2, 5&gt;</p>	C	1	1	1	C	0	1	0	N
N	1	0																											
1	C	1																											
0	1	C																											
N	0	1																											
0	C	1																											
1	1	C																											
C	1	1																											
1	C	0																											
1	0	N																											

**Figure 4.2: Three adjacency matrices of subgraph C-C-N**

It is “conditional” because only when a graph database is given can the canonical adjacency matrix be defined and generated. It is “canonical” because as long as a graph database is given, two isomorphic subgraph patterns must have the same conditional canonical adjacency matrix since two isomorphic subgraph patterns must have the same embeddings and therefore the same lexicographically smallest embedding code.

**DEFINITION 5.2 (Matrix Code).** The matrix code of a subgraph pattern  $p$  is the sequence formed by row-wise concatenation of the lower triangle entries of an adjacency matrix  $M$  of  $p$ .

For example, the matrix codes corresponding to Matrices 1, 2, 3 in Figure 4.2 are N1C01C, N0C11C and C1C10C, respectively.

**DEFINITION 5.3 (CCAM Code).** Given a graph database, where each graph has a unique graph ID, the CCAM Code of a subgraph pattern  $p$  is the matrix code corresponding to the conditional canonical adjacency matrix of  $p$ .

For example, in Figure 4.1, given the graph database composed of  $g_1$  and  $g_2$ , the CCAM code of  $p_1$  is N1C01C.

Previous subgraph pattern encoding methods, such as minimum DFS code [Yan2002] and CAM code [Huan2003], only look at the structural information of the pattern, but do not take advantage of the embedding information. However, in all efficient subgraph pattern mining algorithms, such as FFSM [Huan2003] and SPIN [Huan2004], all embeddings of a pattern are actually already maintained and sorted in increasing order of graph IDs by the algorithms in order to calculate pattern frequency efficiently. Therefore, the embedding information is available when a subgraph pattern mining algorithm computes canonical codes. Given embeddings of a pattern  $p$  in a graph database sorted by graph IDs, the

complexity of computing CCAM code of  $p$  can be reduced to  $O(|V|^2)$ , where  $|V|$  is the number of nodes in  $p$ . The computation can be completed in three steps:

1. Retrieve embeddings with the smallest graph ID
2. For each embedding, sort the node IDs in ascending order and keep track of the lexicographically smallest embedding code B
3. Construct the conditional canonical adjacency matrix according to B and generate the CCAM code

The complexity of the first step can be considered as  $O(1)$  because the embeddings are already sorted and the number of embeddings with the smallest graph ID can be upper-bounded by a small constant in most applications. The complexity of the sorting step is  $O(|V| \lg |V|)$  where  $|V|$  is the number of nodes in  $p$  because the number of embeddings from Step 1 is considered as a constant. The complexity of the third step is  $O(|V|^2)$  because the size of the matrix is  $O(|V|^2)$ . Therefore, CCAM code can be computed in  $O(|V|^2)$  time by taking advantage of embedding information that has been calculated already. This significant improvement in time efficiency is essential to GAIA because GAIA does not require a frequency threshold and therefore cannot prune subgraph patterns based on frequency. Most other subgraph mining algorithms, such as gSpan [Yan2002], FFSM [Huan2003], SPIN [Huan2004], LEAP [Yan2008], gPLS [Saigo2008] and COM [Jin2009], use a frequency threshold to limit the examination to only frequent subgraph patterns. In addition, using CCAM code allows arbitrary edge extensions to a subgraph pattern while previous encoding methods only allows certain types of edge extensions in order to maintain canonical codes of patterns efficiently.

One potential challenge of this encoding method is that the number of embeddings in Step 2 may be large especially when the patterns are small. This problem can be solved by encoding patterns differently according to their sizes: one-edge patterns are encoded with their minimum matrix codes and larger patterns are encoded with their CCAM codes.

## 4.2 Mining Discriminative Subgraph Patterns Using Evolutionary Computation

Evolutionary computation can be viewed as a generic search process for solutions of high quality or fitness, which begins with a set of sample points in the search space and gradually biases to regions of high fitness. In the problem of discriminative pattern mining, discrimination score is used to evaluate the fitness of a subgraph pattern. As a result, our evolutionary search process here is directed toward subgraph patterns with high discrimination power.

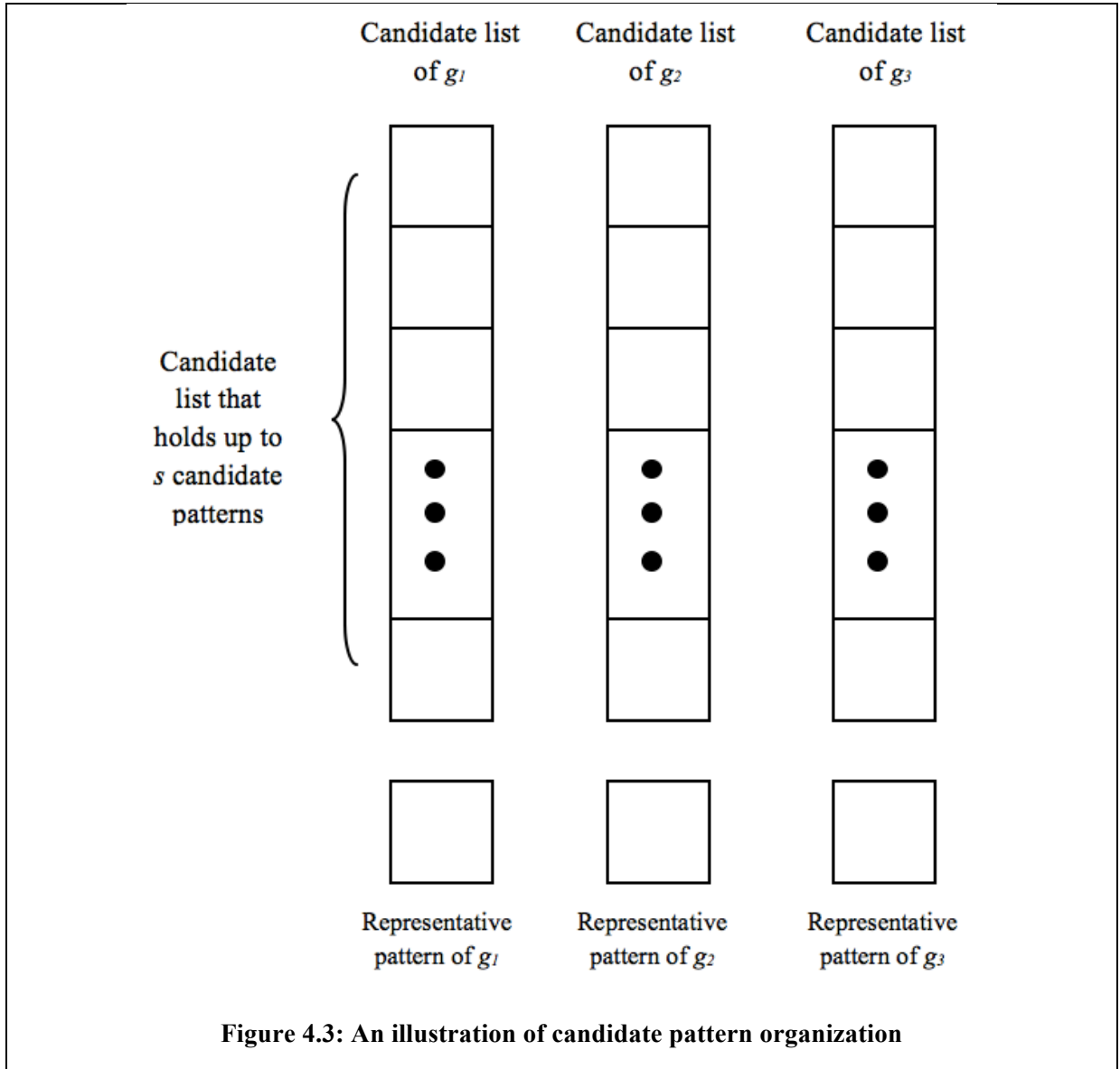
### 4.2.1 Framework of the Pattern Evolution: Organization and Resources

For each graph  $g_i$  in the positive graph set  $G_p$ , the algorithm stores a representative subgraph pattern and a list of up to  $s$  candidate subgraph patterns, where  $s$  is bounded (from above) by the available memory space divided by the number of graphs. Figure 4.3 illustrates the organization of candidate patterns and representative patterns. Only subgraphs of  $g_i$  with discrimination scores greater than 1 can be its representative or in its candidate list. The representative pattern has the highest discrimination score among all patterns that are subgraphs of  $g_i$  found during pattern evolution. Although one pattern can be subgraphs of several positive graphs, each pattern can only be in one candidate list at any time. The candidate lists are initialized with one-edge patterns.

The total number of subgraph patterns that the candidate lists can hold at any time is the product of  $s$  and  $|G_p|$ . This framework is designed to cause selection pressure that can

speed up the convergence of evolutionary search. When the total size of candidate lists is less than the total number of patterns that can be found in positive graphs, not all patterns can be held in the candidate lists at the same time. As a result, candidate patterns need to compete for slots in candidate lists. Generally speaking, the larger the candidate lists are, the less selection pressure and thereby more patterns are considered in the search. When the candidate lists are infinite, the search process becomes an exhaustive search.





Candidate patterns also compete for the opportunity to extend, which is the analog of producing offspring in biological evolution. All subgraph pattern mining algorithms begin with small subgraph patterns that they extend into larger patterns. However, pattern extension is a costly operation and not every pattern extension leads to a discriminative pattern. In an evolutionary search process, candidate patterns compete for the opportunity to extend according to their fitness, which enables the search process to focus on candidate patterns that are more likely to lead to discriminative patterns. Although it does not guarantee that it

reaches the globally optimal solution faster because of the existence of local optimal solutions, our experiments show that in reality it has significant speed advantage over other algorithms.

#### 4.2.2 Pattern Extension

All candidate patterns currently in the candidate lists have a non-zero probability of being selected for pattern extension. To perform pattern evolution, GAIA runs for  $n$  iterations, where  $n$  is a parameter set by the user. During each iteration, GAIA selects one pattern from each candidate list for extension. The probability of pattern  $p$  in candidate list of  $g_i$  to be selected for extension is proportional to the log ratio score of  $p$  and is calculated as follows:

$$Probability(p \text{ is selected}) = \frac{score(p)}{\sum_{p' \text{ is in the candidate list of } g_i} score(p')}$$

The probability is always between 0 and 1 because only patterns with positive log ratio scores are allowed in candidate lists as described in Subsection 3.2. This selection method is commonly used in evolutionary algorithms and an analysis on it can be found in [De Jong2006]. The intuition here is that candidate patterns with higher scores are more likely to be extended to patterns with high scores because structurally similar subgraph patterns have similar discrimination power [Yan2008]. Note that when  $s = 1$ , each candidate list only holds 1 pattern. The probability of this pattern being selected for extension is 1. When  $s > 1$ , multiple patterns may be held in a candidate list. A random number generator is used to determine which pattern is selected for extension according to their probabilities.

For an extension operation of pattern  $p$ , GAIA generates a pattern set  $X(p)$  and each pattern  $p'$  in  $X(p)$  has one new edge attached to  $p$ . This new edge is not present in  $p$  and it can

be either between two existing nodes in  $p$  or between one node in  $p$  and a new node. Unlike many previous subgraph pattern mining algorithms that only extend patterns with certain types of edges in order to efficiently maintain their canonical codes, GAIA considers all one-edge extensions of pattern  $p$  that occur in the positive graphs. This difference in extension operation is essential to GAIA because evolutionary computation is essentially a heuristic search for optimal solution. This difference enables GAIA to explore the candidate pattern space in any direction that appears promising.

Extensions of different patterns can produce the same pattern because a pattern  $p$  with  $k$  edges can be directly extended from all of its subgraphs with  $k-1$  edges. Therefore, a lookup table is needed by GAIA to determine whether a pattern has already been generated to avoid repetitive examination of the same pattern. The codes for pattern lookup are generated by the encoding method described in Chapter 4.1.

### 4.2.3 Pattern Migration and Competition

In most cases, an extension operation on one pattern generates many new patterns and as a result the number of patterns found by the algorithm grows. Sooner or later the number of patterns will exceed the number of available positions in the candidate lists. It is also possible that the number of one-edge patterns already exceeds the number of available positions in the candidate lists at the very beginning if  $s$  is small. Therefore some rules are needed to determine which patterns should survive in the candidate lists and which candidate list they should dwell in.

First, a pattern that has already been extended should not “live” in the candidate lists any longer because it has served its role in generating new patterns.

Second, some pattern in the candidate list may migrate to the candidate list of another graph if such migration will increase its chance of survival. Let  $p$  be the candidate pattern for migration and  $G(p)$  be the set of graphs containing  $p$ . Let  $g_i$  be the graph in  $G(p)$  which has the lowest value of  $\sum_{p' \text{ is in the candidate list of } g_i} \text{score}(p')$ .  $p$  will migrate to the candidate list of  $g_i$ . The rationale for this pattern migration is that if a pattern wants to survive then it should go to a candidate list with the least fierce competition. In GAIA, the fierceness of competition of a candidate list is measured by the sum of scores of patterns in the list.

If the candidate list of  $g_i$  still has vacant positions, then  $p$  can move into one vacant position directly. However, if the candidate list is already full, then  $p$  has to compete with the “resident” patterns in the list. One straightforward algorithm to let  $p$  compete with “resident” patterns is to compare the log ratio score of  $p$  and the minimum log ratio score among “resident” patterns. If the score of  $p$  is greater than the minimum score among “resident” patterns, then  $p$  takes the position of pattern  $p'$  with the minimum score and  $p'$  no longer exists in any candidate list; otherwise,  $p$  fails to survive and will not exist in any candidate list. The disadvantage of this greedy algorithm is that it ignores the fact that patterns with low log ratio scores may still have some potential to extend into patterns with high log ratio scores and patterns with high log ratio scores at the time may have reached their limits and will never extend to better patterns. Therefore, GAIA adopts a randomized method for pattern competition which is commonly used by evolutionary algorithms. The score of  $p$  is compared against the score of a pattern  $p'$ , which is randomly selected with probability  $1/s$  from the candidate list. If the score of  $p$  is higher, then  $p'$  is eliminated and  $p$  takes the position of  $p'$ ; otherwise,  $p$  is eliminated. By doing so, GAIA can at least have a chance to protect some of the “weak” patterns and give them an opportunity to extend into “strong”

patterns. The benefit of this randomized algorithm is more evident when  $s$  is reasonably large. Note that when  $s = 1$  the randomized strategy is essentially the same as the greedy strategy.

Again, the exhaustive extension operation is of great importance to allow pattern competition and elimination. When GAIA eliminates a pattern  $p$ , the real loss is not only this pattern but also the patterns generated by extending  $p$ . In previous subgraph pattern mining algorithms, such as gSpan [Yan2002] and FFSM [Huan2003], a pattern  $p$  can only be extended from one of its subpatterns,  $p'$ . If  $p'$  is lost, then the algorithms will never find  $p$ . As a result, for these algorithms, allowing pattern elimination will surely lose many patterns, some of which are discriminative patterns. But in GAIA, eliminating  $p'$  does not necessarily lead to the loss of  $p$  because the exhaustive extension operation allows  $p$  to be extended from many different patterns. As a result, the risk of missing discriminative patterns is much lower than other subgraph mining algorithms.

The algorithms for pattern migration and pattern evolution are described as follows.

**Algorithm:** *Pattern\_Migrate* ( $p, T$ )

$p$ : a pattern

$T$ : candidate lists

1.  $g = \text{argmin}_{g'} (\sum_{p \text{ is in the candidate list of } g'} \text{score}(p))$
2. **if** (the candidate list of  $g$  has vacant positions)
3.   insert  $p$  into the candidate list of  $g$
4. **else**
5.   randomly select a pattern  $p'$  in the candidate list of  $g$
6.   **if** ( $\text{score}(p) > \text{score}(p')$ )
7.     replace  $p'$  with  $p$

**Algorithm:** *Pattern\_Evolution* ( $G_p, G_n, n = \text{INT\_MAX}, s = \text{available\_space}/|G_p|$ )

$G_p$ : positive graph set

$G_n$ : negative graph set

$s$ : maximum size of each candidate list, by default equal to  $\text{available\_space}/|G_p|$

$n$ : maximum number of iterations, by default the maximum interger value in the system

```

T: all candidate lists
H: lookup table of patterns that have already been found
D = {all edges that occur in  $G_p$ }
1. for each edge  $e$  in  $D$ 
2.   Pattern_Migrate ( $e, T$ )
3. for  $k = 1:n$ 
4.   if (all candidate lists are empty)
5.     break
6.   for each  $g$  in  $G_p$ 
7.     randomly select a pattern  $p$  in the candidate list of  $g$ 
8.      $X(p) = \{\text{all patterns in } G_p \text{ with one more edge attached to } p\}$ 
9.     for each pattern  $p'$  in  $X(p)$ 
10.      if (CCAM code of  $p'$  is in  $H$ )
11.        continue
12.      insert  $p'$  into  $H$ 
13.      Migrate ( $p', T$ )
14.      update representative patterns

```

#### 4.2.4 Generating Consensus Results

Because GAIA is a randomized algorithm (when  $s > 1$ ), each single run of pattern evolution may generate different representative patterns and consume varying amount of CPU time. Some runs of pattern evolution may find better representative patterns than others and thus lead to classifiers with higher normalized accuracy. Therefore, if GAIA runs many instances of pattern evolution in parallel and selects the best subgraph patterns from all representative patterns found by these instances of pattern evolution, it is likely that GAIA can get a better set of discriminative subgraph patterns than using representative patterns from one instance of pattern evolution alone. Therefore, by generating a consensus model based on many parallel instances of pattern evolution and only using the fastest instances of pattern evolution, GAIA can improve the discrimination power of its results and achieve faster expected response by taking advantage of parallel computing, which cannot be done easily in other discriminative subgraph mining algorithms.

### 4.3 Experiments

I analyzed the performance from two perspectives: runtime efficiency and normalized accuracy in graph classification applications. I evaluated two versions of GAIA: single-GAIA (when  $c=1$ ) only performs one instance of pattern evolution to mine discriminative patterns and parallel-GAIA (when  $c>1$  and  $s>1$ ) runs in parallel  $c$  instances of pattern evolution, where  $c$  is a user-specified parameter, to mine discriminative patterns.

For each experiment, I ran GAIA (for both single-GAIA and parallel-GAIA) 5 times and report the average normalized accuracy and average runtime of the 5 runs. Note that GAIA is a randomized algorithm and each run may have slightly different classification accuracy and runtime even though the standard deviations are very small. For chemical datasets, standard deviations of normalized accuracies are less than 0.01 and standard deviations of runtimes are usually less than 1 second for single-GAIA and less than 0.1 second for parallel-GAIA. For protein datasets, standard deviations of normalized accuracies are usually less than 0.03 and standard deviations of runtimes are less than 0.1 seconds. Therefore, I only reported the average in the following analysis.

#### 4.3.1 GAIA Performance Analysis

In this section, I studied the performance of GAIA with respect to three parameters:  $s$  (maximum number of positions in a candidate list),  $n$  (maximum number of iterations) and  $c$  (number of instances of pattern evolution).

First, I ran single-GAIA ( $c = 1$ ) with different  $s$  and  $n$  on the unbalanced chemical datasets and showed the average normalized accuracy and average runtime in Table 4.1 and Table 4.2. In Table 4.1,  $n$  is fixed at 4 and it can be seen that the normalized accuracy is generally insensitive to the variation in  $s$ . When  $n$  is large enough, larger  $s$  enables GAIA to

perform a more extensive search for discriminative patterns because it allows more candidate patterns to be stored in candidate lists and visited in the search process, which is why when the value of  $s$  increases from 1 to 7 the normalized accuracy also increases. When the value of  $s$  further increases, the normalized accuracy starts to decrease because although the size of candidate lists allows GAIA to perform a more exhaustive search, GAIA only runs for  $n$  iterations and thus fails to take advantage of the large candidate lists. It can also be seen that although  $n$  is fixed, the average runtime varies and is correlated with the normalized accuracy. This is because, generally speaking, the more discriminative a pattern is the more frequent and larger it is and thus the more time it takes to compute all of its embeddings and perform extensions. In Table 4.2, I fixed  $s$  at 10 and studied the performance of single-GAIA with respect to  $n$ . I observed that increasing  $n$  can effectively improve normalized accuracy when  $n$  is small but only has marginal effect when  $n$  is large. When  $n$  is small, pattern evolution is far from convergence after  $n$  iterations and larger  $n$  can make the result closer to convergence. When  $n$  is sufficiently large, pattern evolution is already near convergence and further increase in  $n$  has little effect. Similarly, Table 4.2 also shows that larger  $n$  results in longer runtime due to more iterations. I provided  $n$  as an optional parameter for applications in which speed is crucial.

**Table 4.1: Normalized accuracy and average runtime of single-GAIA with different values of  $s$ , where  $n = 4$  (chemical datasets)**

$s$	Normalized accuracy	Average runtime (sec)
1	0.7295	2.3398
3	0.7329	2.7545
5	0.7310	2.8725
7	0.7330	2.8705
10	0.7298	2.7444
30	0.7311	2.4278



50	0.7300	2.4080
70	0.7293	2.4207

**Table 4.2: Normalized accuracy and average runtime of single-GAIA with different values of  $n$ , where  $s = 10$  (chemical datasets)**

$n$	Normalized accuracy	Average runtime (sec)
1	0.7050	1.4192
2	0.7198	1.8795
4	0.7320	2.8066
8	0.7325	4.0611
16	0.7363	5.7075
32	0.7368	8.8772

Then I studied the effect of  $c$  to the performance of GAIA. Figure 4.4 and Figure 4.5 show the average normalized accuracies of running GAIA with different  $c$  on chemical datasets and protein datasets respectively. Both figures illustrate that increasing  $c$  can result in higher average normalized accuracy. This positive correlation is due to the randomization in pattern evolution. Each instance of pattern evolution finds different representative patterns. One instance may be able to find a good representative pattern for  $g_i$  but fail to find one for  $g_j$  while another instance returns a good pattern for  $g_j$  but not for  $g_i$ . Therefore when the representative patterns from different instances are merged together, the average quality of representative patterns can be improved. Generally, the larger the value of  $c$ , the better the normalized accuracy of GAIA. Figure 4.6 and Figure 4.7 show the average runtime of GAIA with different  $c$  on chemical datasets and protein datasets respectively. Both figures show the same trend of average runtime as  $c$  increases: runtime starts to converge when  $c$  is large. The runtime of GAIA is the sample median of the running times of  $c$  pattern evolution instances. When  $c$  is large enough, the runtime (sample median) should converge to the theoretical

median of the runtime of all possible pattern evolution instances. Therefore, larger  $c$  leads to more stable runtime.

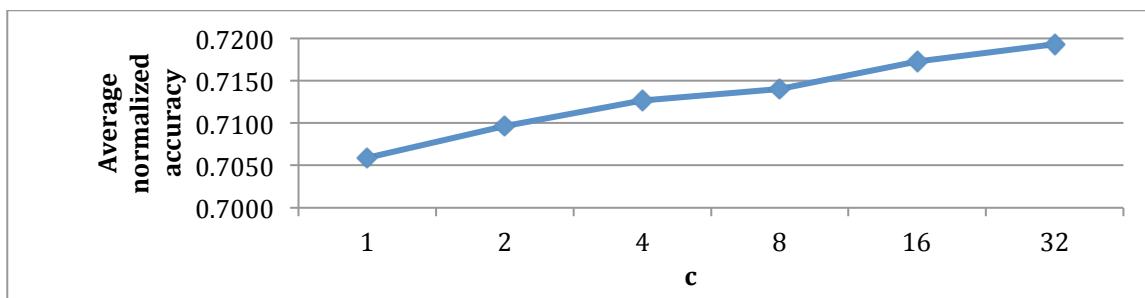


Figure 4.4: average normalized accuracy vs.  $c$  (chemical datasets)

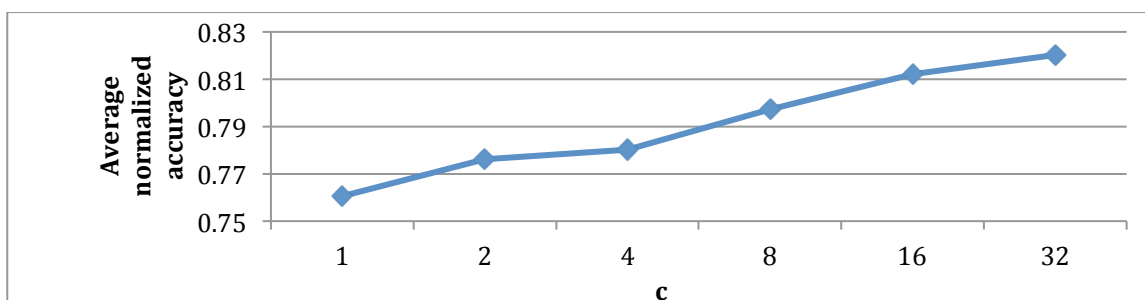


Figure 4.5: average normalized accuracy vs.  $c$  (protein datasets)

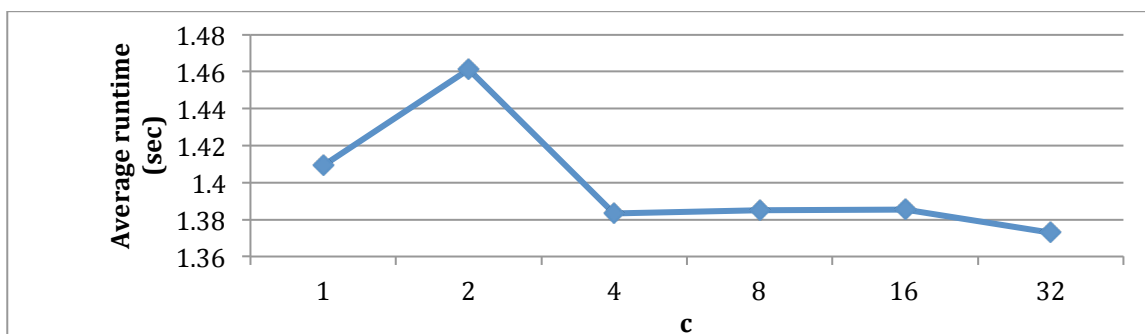


Figure 4.6: average runtime vs.  $c$  (chemical datasets)

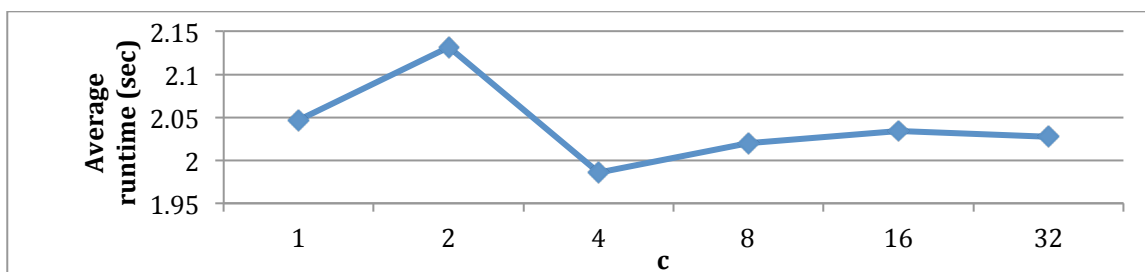
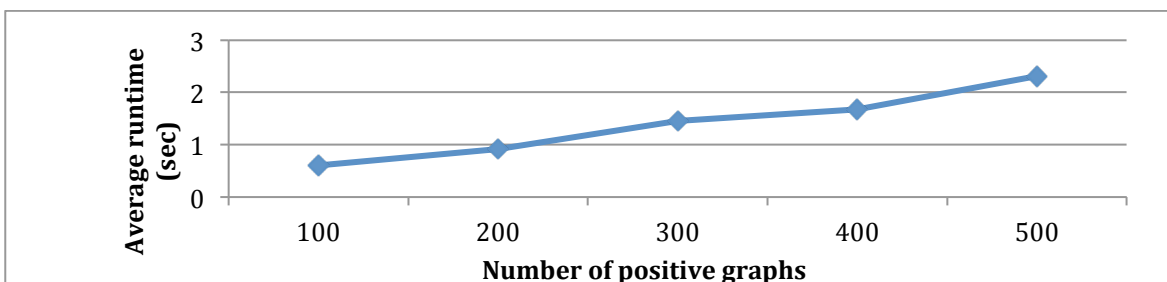
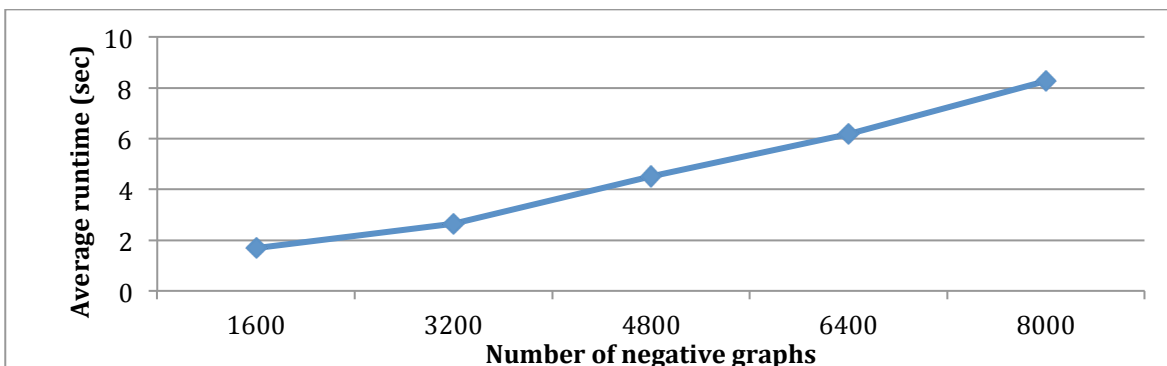


Figure 4.7: average runtime vs.  $c$  (protein datasets)

Figure 4.8 and Figure 4.9 demonstrate the scalability of GAIA for chemical datasets as the number of positive graphs and number of negative graphs increase. In Figure 4.8, the number of negative graphs is fixed at 1600 and the number of positive graphs varies. The average runtime grows approximately linearly as the number of positive graphs increases. In Figure 4.9, the number of positive graphs is fixed at 400 and the number of negative graphs varies. It can be seen that the average runtime is linear to the number of negative graphs.



**Figure 4.8: average runtime vs. number of positive graphs (chemical datasets, number of negative graphs=1600)**



**Figure 4.9: average runtime vs. number of negative graphs (chemical datasets, number of positive graphs=400)**

## 4.3.2 Comparison with Other Methods

### 4.3.2.1 Parallel-GAIA

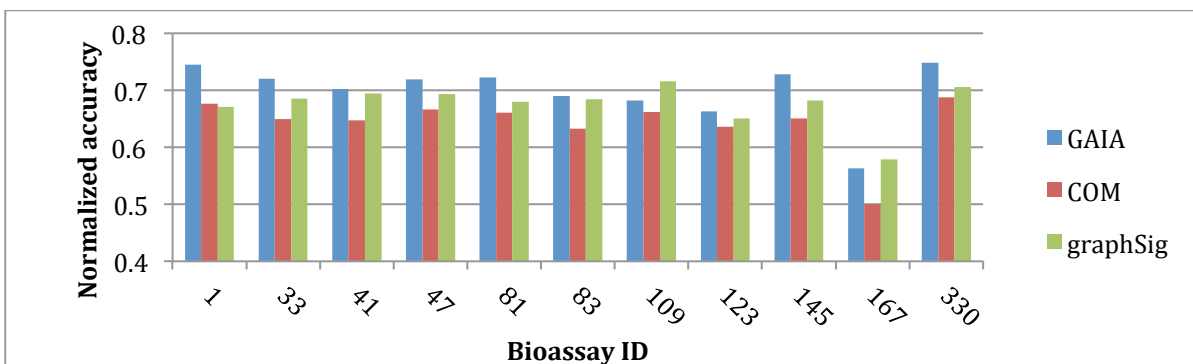
This section compares parallel-GAIA (denoted as GAIA in this section for simplicity) with two other state-of-the-art graph classification methods: COM [Jin2009] and graphSig [Ranu2009], both of which claim to outperform LEAP [Yan2008] in terms of runtime with

competitive classification accuracy. COM also outperforms gPLS for the protein datasets. For GAIA, I used the parameters that give the best trade-off between runtime efficiency and classification accuracy. For COM and graphSig, I set the parameters that deliver the best results as suggested in [Jin2009] and [Ranu2009] respectively. The parameters for the three methods are summarized in Table 4.3. To compare with graphSig, I used the eleven chemical datasets and randomly sample 400 actives and 400 inactives from each dataset to form the training sets, because graphSig is implemented for balanced datasets. I also generated balanced protein datasets (number of negative graphs = number of positive graphs) to evaluate graphSig.

**Table 4.3: Summary of parameters**

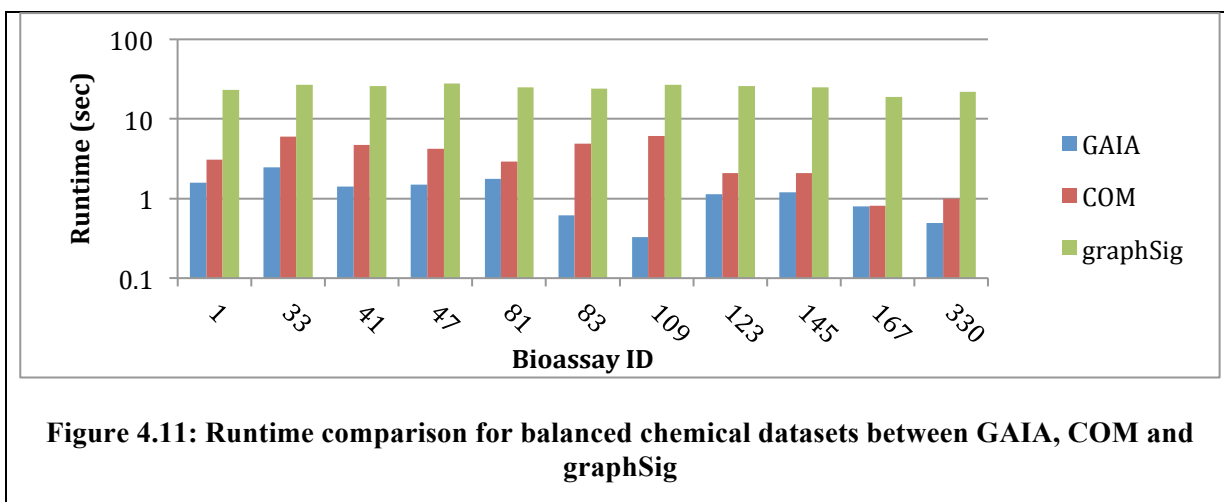
	Parameters for chemical datasets	Parameters for protein datasets
GAIA	$s = 10, n = 4, c = 32$	$s=100, n=10, c=32$
COM	$t_p=1\%, t_n=0.4\%$	$t_p=30\%, t_n=0\%$
graphSig	$\text{maxPvalue}=0.1, \text{minFreq}=0.1\%$	$\text{maxPvalue}=0.1, \text{minFreq}=0.1\%$

Figure 4.10 shows the normalized accuracy comparison between graphSig, GAIA and COM for balanced chemical datasets. GAIA delivers better normalized accuracy than graphSig on most datasets though not all of them. The average normalized accuracy of GAIA is 2.2% higher than that of graphSig. COM generally has lower normalized accuracy than GAIA and graphSig.



**Figure 4.10: normalized accuracy comparison for balanced chemical datasets between GAIA, COM and graphSig**

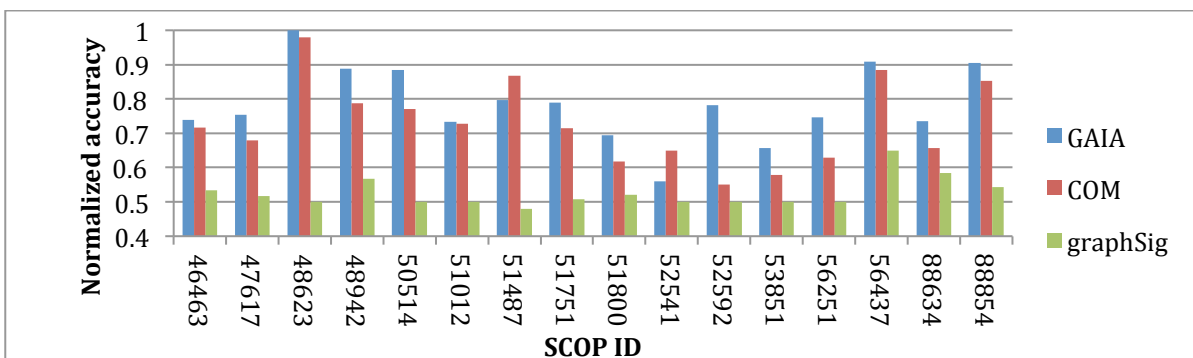
Figure 4.11 compares the runtime of graphSig, GAIA and COM for the balanced chemical datasets. GAIA demonstrates a huge advantage in runtime performance. For all datasets, GAIA is 20.44 times faster than graphSig on average. In addition, GAIA also outperforms COM considerably for every chemical dataset in terms of speed (on average 2.83 times faster).



**Figure 4.11: Runtime comparison for balanced chemical datasets between GAIA, COM and graphSig**

Figure 4.12 and Figure 4.13 compare the normalized accuracy and average runtime performance respectively between graphSig, GAIA and COM. GraphSig is not comparable in processing protein datasets. Its speed is 2 orders of magnitude slower and its normalized accuracy is close to 0.5. This is mainly because graphSig is specifically designed and

optimized for chemical compound datasets. Between GAIA and COM, GAIA is on average about 1.2 times faster and has a normalized accuracy 5.7% higher than that of COM.

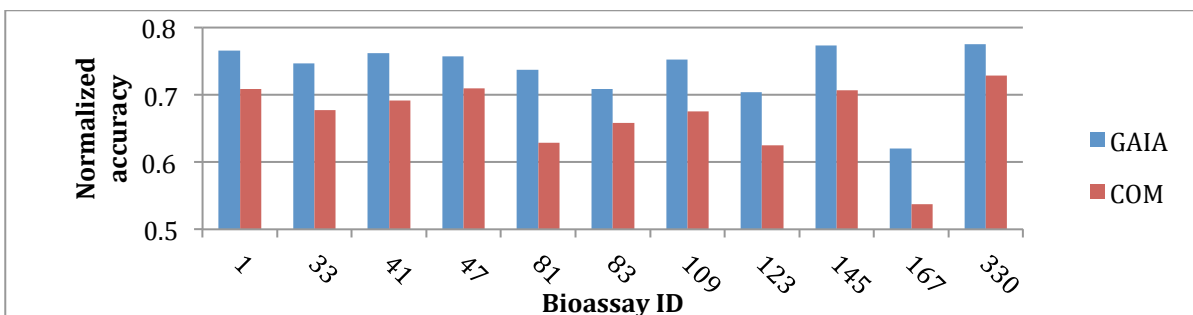


**Figure 4.12: Normalized accuracy comparison for balanced protein datasets between GAIA, COM and graphSig**

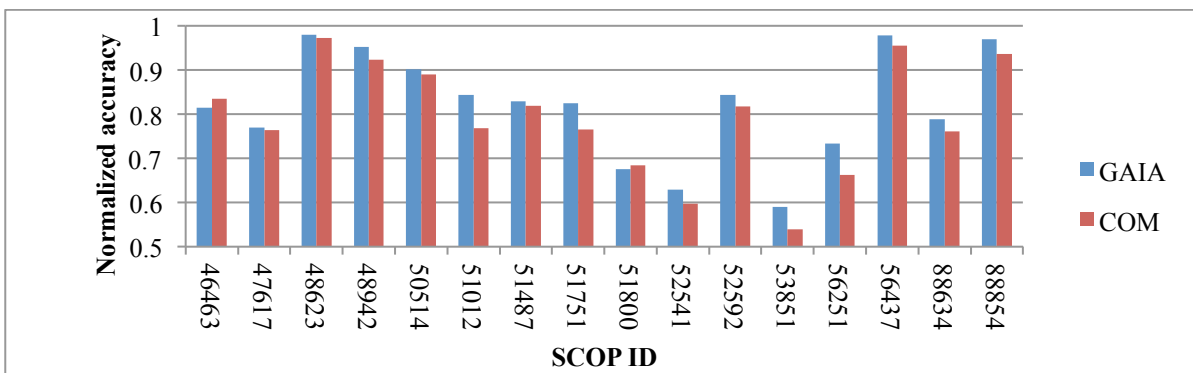


**Figure 4.13: Runtime comparison for balanced protein datasets between GAIA, COM and graphSig**

Since both GAIA and COM can handle unbalanced datasets, I used the unbalanced chemical datasets and unbalanced protein datasets described at the beginning of this section to provide additional comparison. Figure 4.14 and Figure 4.15 show the normalized accuracy comparison. For chemical datasets, normalized accuracy of GAIA is 6.86% higher than that of COM on average. For protein datasets, normalized accuracy of GAIA is 2.7% higher than that of COM on average.



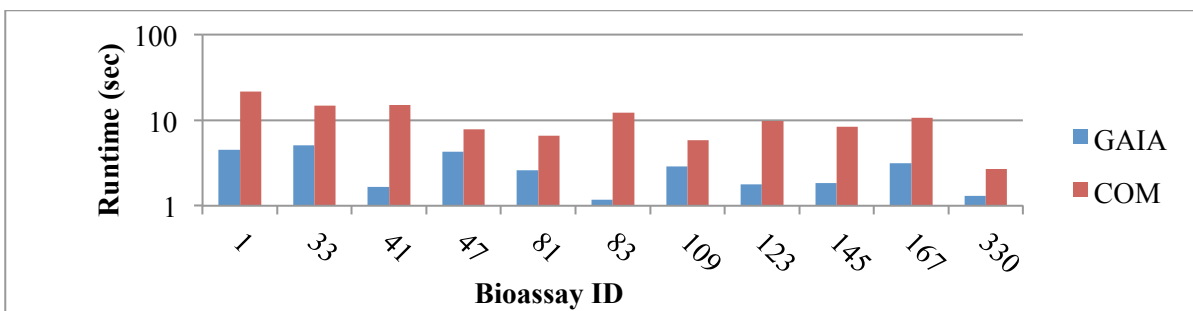
**Figure 4.14: Normalized accuracy comparison for unbalanced chemical datasets between GAIA and COM**



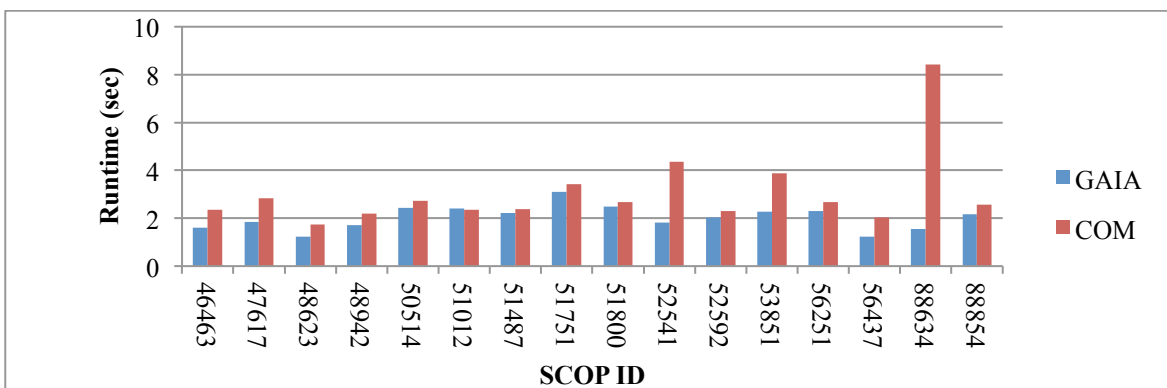
**Figure 4.15: Normalized accuracy comparison for protein datasets between GAIA and COM**

Figure 4.16 and Figure 4.17 compare the runtimes of the two methods for chemical datasets and protein datasets respectively. On average, GAIA is 3.8 times faster than COM for chemical datasets and 1.5 times faster than COM for protein datasets.





**Figure 4.16: runtime comparison for unbalanced chemical datasets between GAIA and COM**



**Figure 4.17: runtime comparison for unbalanced protein datasets between GAIA and COM**

#### 4.3.2.2 Single-GAIA

Considering that parallel-GAIA demands more computation resources than COM and graphSig when outperforming them, I also compared single-GAIA with the other two methods to further demonstrate the advantage of GAIA. Table 4.4 summarizes the average runtime and average normalized accuracy comparisons between single-GAIA, parallel-GAIA, COM and graphSig. The parameters are the same as listed in Table 4.3 except that for single-GAIA  $c=1$ . It can be seen that even without parallel computing, single-GAIA can excel COM and graphSig in terms of both average runtime and normalized accuracy. The only exception is that the average normalized accuracy of single-GAIA for the unbalanced protein datasets is 3.27% lower than that of COM.

**Table 4.4: summary of comparison between single-GAIA, parallel-GAIA, COM and graphSig**

		Single-GAIA	Parallel-GAIA	COM	graphSig
Balanced chemical datasets	Runtime (sec)	1.296	1.210	3.430	24.73
	Accuracy	0.7029	0.6988	0.6428	0.6768
Balanced protein datasets	Runtime (sec)	0.5996	0.5748	0.6788	51.13
	Accuracy	0.7665	0.7855	0.7285	0.5250
Unbalanced chemical datasets	Runtime (sec)	2.807	2.752	10.44	N/A
	Accuracy	0.7320	0.7368	0.6682	N/A
Unbalanced protein datasets	Runtime (sec)	2.047	2.028	3.059	N/A
	Accuracy	0.7605	0.8202	0.7932	N/A

## Chapter 5

### Mining Discriminative Subgraph Patterns by Learning from Search History

Discriminative subgraph pattern mining can be considered as an optimization problem with a user-specified score function, whose search space includes all possible subgraphs. This search problem is typically solved in one of two ways: one is a greedy approach attempting to reach local optimal subgraph(s) as fast as possible; the other is a branch-and-bound approach that prunes the search space using an estimated upper-bound of the scores. I proposed a new discriminative subgraph pattern mining algorithm, named LTS (Learn To Search) [Jin2011], which integrates both approaches with novel probing and pruning techniques.

A tight estimated upper-bound of scores enables a branch-and-bound algorithm to prune branches that a loose estimated upper-bound is unable to and thereby leads to a smaller search space. COM [Jin2009] and GAIA [Jin2010] compute a very loose estimated upper-bound by assuming a constant positive frequency and zero negative frequency of “descendant” patterns in the subgraph enumeration tree. LEAP [Yan2008] proposes the prune-by-structural-proximity strategy, which is based on an observation that subgraph patterns with similar structures tend to have similar scores. It allows LEAP to calculate a tighter estimated upper-bound than COM and GAIA. I discovered that a tight estimated upper-bound can also be achieved by learning from search history. I characterized a pattern by a sequence of scores of the “ancestor” patterns visited on the path to the pattern and I refer

to this sequence as the score record of the pattern. I first sampled the search space using a probing algorithm that locates a few discriminative subgraphs and acquires their score records. Then I explored the search space in a branch-and-bound fashion, using the score records to estimate an upper bound of scores along an exploration branch. When I computed the upper-bound for “descendants” of pattern  $p$ , I looked for a pattern  $q$  probed during the first phase whose score record is the same as that of  $p$  and use the observed upper-bound of  $q$  as the estimated upper-bound for  $p$ . To facilitate this idea, the probing algorithm needs to be able to find subgraph patterns with high scores in a short amount of time. If the probing algorithm is unable to reach subgraph patterns with high discrimination scores, then the upper-bound based on the sample subgraphs from the probing algorithm tend to be underestimated and thus the optimal subgraph pattern may be missed. If the probing algorithm is very slow, then even if it finds subgraphs with high scores, it is still unable to improve the overall efficiency of the search. I proposed an efficient probing algorithm that satisfies both requirements and also approaches the optimal score fast.

In a branch-and-bound algorithm that looks for the most discriminative subgraph(s), if the estimated upper-bound of scores along a search branch is not greater than the optimal score found so far, then the branch can be pruned without the risk of missing the optimal solution. Therefore, the higher the optimal score found so far, the more the pruned search space. Note that the optimal score found so far is always less than or equal to the true optimal score in the search space. So approaching the optimal score faster leads to more efficient pruning. LEAP achieves this goal by frequency-descending mining, which searches for discriminative subgraph patterns with high frequencies first to find a high score and then searches again with the help of the high score to prune. This search technique is based on an

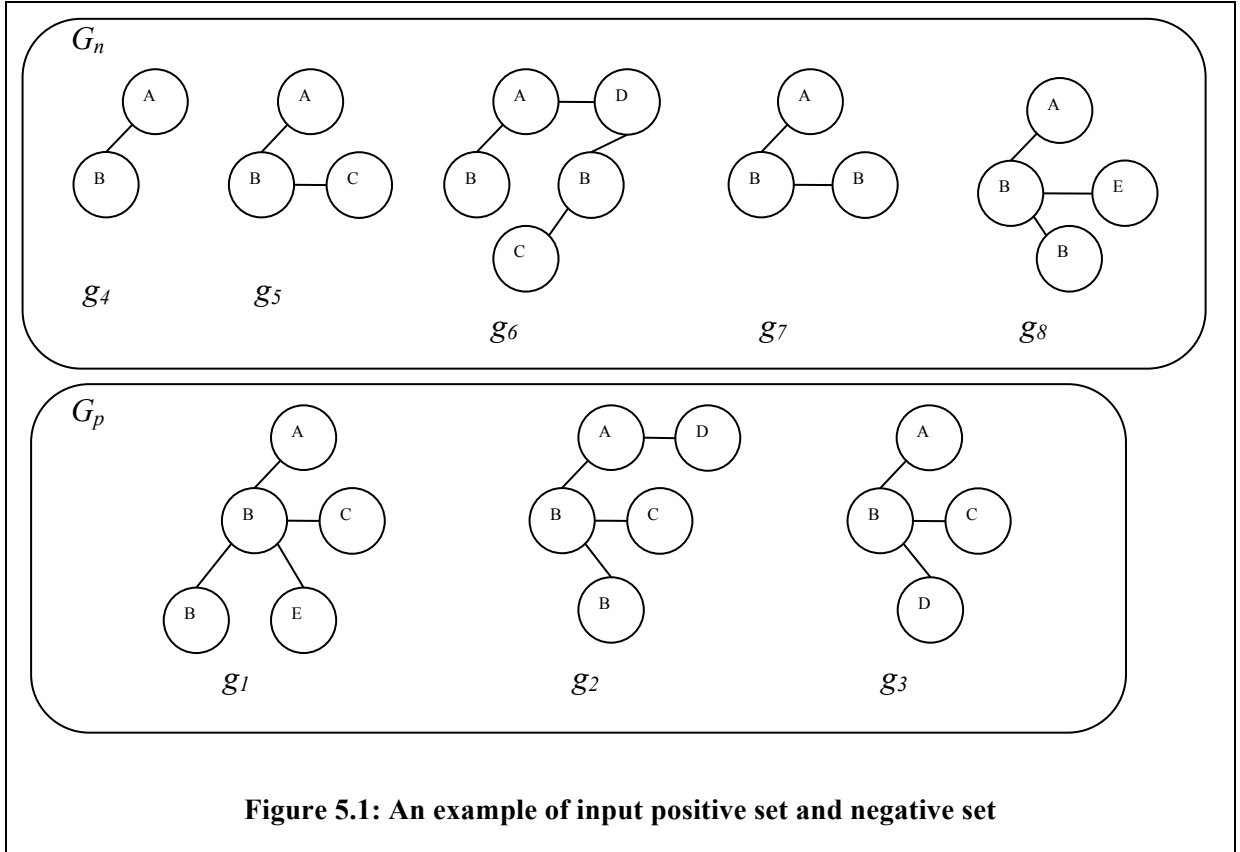
observation that subgraphs with high frequencies are more likely to have high scores. COM approaches the optimal score faster by exploring candidate subgraph patterns in the order of their changes in scores from their ancestors, which leverages the observation that patterns that have dramatic increase in scores are very likely to lead to patterns with even higher scores. GAIA utilizes evolutionary computation to approach the optimal score even faster, based on the assumption that patterns with higher scores are more likely to lead to the optimal pattern. In this chapter, a greedy probing algorithm, namely *fast-probe*, is proposed to efficiently locate sample subgraphs with high scores and compute their score records. These subgraphs and score records are used to achieve efficient pruning in the subsequent branch-and-bound search for discriminative subgraph patterns. *Fast-probe* only preserves and extends the best candidate subgraph patterns discovered so far. It uses multiple-lineage exploration instead of single-lineage exploration to compensate for its aggressiveness in pruning patterns and allow it to still find patterns with high discrimination scores.

The overall proposed mining algorithm LTS is a two-step method. First it invokes *fast-probe* to acquire search history and then performs a branch-and-bound search which utilizes the optimal scores from *fast-probe* as a starting point and estimates upper-bound based on search history. Experimental results show that *fast-probe* alone is a competitive discriminative subgraph mining algorithm compared with state-of-the-art competitors, and moreover, LTS as a whole can find even better subgraph patterns without prolonged runtime.

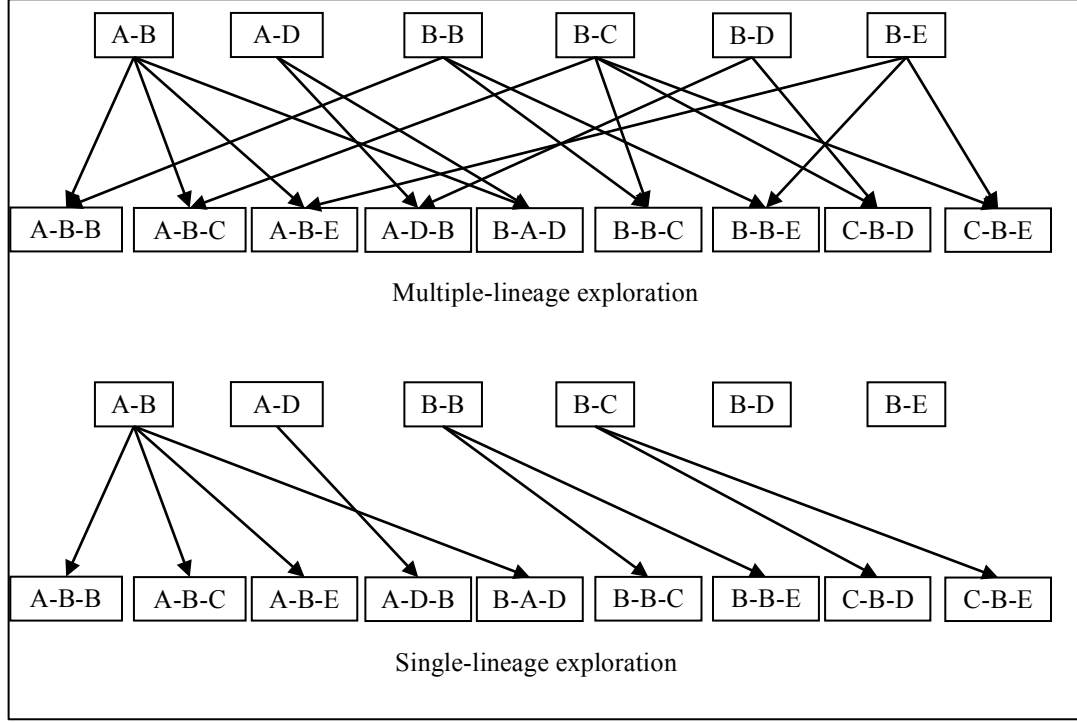
## 5.1 Pattern Exploration Order

Almost all efficient subgraph pattern exploration methods, such as gSpan [Yan2002] and FFSM [Huan2003], start with subgraphs having only one edge and extend them to larger subgraphs by adding one edge at a time. Each large subgraph pattern can be directly extended

from more than one smaller subgraph patterns. For example, in Figure 5.1, subgraph pattern  $A-B-C$  can be extended from either  $A-B$  or  $B-C$ .



**DEFINITION 6.1 (Lineage).** In a pattern exploration method  $M$ , a lineage of pattern  $p$  is a sequence of patterns:  $l(p) = p_1 p_2 \dots p_{k-1} p_k$ , where  $p_k = p$ ,  $p_1$  has only one edge and  $\forall i \in [1, k-1]$ ,  $p_{i+1}$  can be directly extended from  $p_i$  by adding one more edge.



**Figure 5.2: An example of multiple-lineage exploration and single-lineage exploration for graphs in Figure 5.1**

If a pattern exploration method allows a pattern to have multiple lineages, I call this exploration method a multiple-lineage exploration; otherwise, I call it a single-lineage exploration. Figure 5.2 shows examples of multiple-lineage exploration and single-lineage exploration for the graph sets in Figure 5.1. Each node represents a subgraph pattern (only patterns with less than 3 edges are shown for illustration) in the graph sets and there is a directed edge from node  $p$  to node  $q$  if in the exploration method pattern  $q$  is allowed to be reached by extending  $p$ .

A subgraph pattern may have multiple possible lineages. Thus, multiple-lineage exploration is more natural than single-lineage exploration. To achieve single-lineage exploration, an algorithm needs to define an enumeration order  $<$  on all subgraph patterns in the search space. If pattern  $p <$  pattern  $q$ , then  $p$  is enumerated before  $q$ . The resulting

lineages become the canonical lineages of the respective patterns. Both gSpan and FFSM are single-lineage exploration methods.

The major advantage of single-lineage exploration is that it is more efficient than multiple-lineage exploration in subgraph pattern enumeration without missing any pattern. In a single-lineage exploration method, each subgraph pattern is enumerated only once while a multiple-lineage exploration method may visit a pattern multiple times through different lineages. For example, in Figure 5.2, the multiple-lineage exploration visits pattern  $A-B-B$  twice while the single-lineage exploration visits it only once. In addition, the average number of subgraph extensions performed for each subgraph pattern in single-lineage exploration is less than that in multiple-lineage exploration. For example, in Figure 5.2, each subgraph pattern with one edge performs three extension operations on average in multiple-lineage exploration while the average number of extension operations in single-lineage exploration is 1.5. Extension operation is the most costly operation in subgraph enumeration, thus algorithms requiring fewer extensions are highly favorable. In applications where subgraph patterns are much larger and more complex, the difference in number of extension operations becomes even larger. As a result, single-lineage exploration is preferred in most subgraph mining algorithms.

However, single-lineage exploration has the problem that its result is sensitive to subgraph pruning. Since each subgraph pattern can only be reached through a single lineage, the algorithm will miss a subgraph pattern if any subgraph on its lineage is pruned. On the contrary, multiple-lineage exploration is much more tolerant of subgraph pruning because a subgraph pattern can be reached through more than one lineage. This difference does not create any problem for using single-lineage exploration in frequent subgraph mining because



of the antimonotonicity property of pattern frequency. In frequent subgraph mining, if pattern  $p$  is in the lineage of pattern  $q$  and  $q$  is a frequent pattern, then  $p$  must also be frequent by the mining algorithm. However, in discriminative subgraph pattern mining, the redundancy in multiple-lineage exploration becomes its advantage over single-lineage exploration. Objective functions to measure discrimination power of subgraphs are usually not antimonotonic. If pattern  $p$  is in the lineage of pattern  $q$  and  $q$  is a discriminative pattern,  $p$  is not necessarily discriminative. Under such circumstances, multiple-lineage exploration can be aggressive in pruning patterns with low discrimination scores while single-lineage exploration cannot afford to prune any pattern unless it is absolutely certain that the pattern will not lead to any discriminative pattern.

For example, in Figure 5.1,  $A-B-C$  is a highly discriminative subgraph pattern in the positive set while  $A-B$  is not discriminative as it appears in every positive and negative graph. The single-lineage exploration shown in Figure 5.2 cannot prune  $A-B$  because otherwise  $A-B-C$  will be missed. The multiple-lineage exploration in Figure 5.2 can afford to prune  $A-B$  since  $A-B-C$  can also be reached from  $B-C$ .

In the proposed algorithm, LTS, I adopted multiple-lineage exploration to reduce the risk of missing the most discriminative subgraph patterns due to pruning. I used CCAM code to encode subgraph patterns and maintain a lookup table for subgraph patterns that have been extended to avoid extending a subgraph pattern repeatedly. Embeddings of subgraph patterns in the graph sets are also maintained to facilitate subgraph extension and frequency calculation.

## 5.2 Fast Probing Subgraph Pattern Space

As mentioned earlier in this chapter, a greedy algorithm can often reach a (relatively) discriminative subgraph quickly. Even though it may not be the optimal one, its score can be used to prune the search space. The higher the score, the better the pruning power. For example, let the estimated upper-bound for descendants of  $p$  be 1.0. By the time  $p$  is visited, if the best score so far is 1.2, all descendants of  $p$  can be pruned. But if the best score found so far is only 0.5, the algorithm is unable to perform any pruning.

I proposed a greedy algorithm called *fast-probe* to generate a good sample of discriminative subgraphs to facilitate the subsequent branch-and-bound search. *Fast-probe* maintains a list of candidate subgraph patterns to be processed. The candidate list is initialized with all single-edge subgraph patterns in  $G_p$ . It repeatedly draws and processes a candidate pattern  $p$  from the list as long as the list is not empty. If pattern  $p$  is the optimal pattern for any positive graph at the time it is processed, *fast-probe* computes all extensions of  $p$  in the positive set with one more edge and put an extension into the candidate list if the extension has not be generated before; otherwise,  $p$  is discarded. *Fast-probe* terminates when the candidate list becomes empty. This process is efficient since only the best subgraphs are extended to generate candidate patterns.

The algorithm is described below.

**Algorithm:** *fast-probe* ( $G_p, G_n$ )

$G_p$ : positive graph set

$G_n$ : negative graph set

*Candidate\_list*: the set of subgraph patterns to be extended

1. Put all single-edge subgraph patterns into *candidate\_list*
2. while (*candidate\_list* is not empty)
3.    $p \leftarrow$  get next pattern and remove it from *candidate\_list*
4.    $updated \leftarrow false$
5.   for each graph  $g$  in  $G_p$
6.     if  $score(p) > \text{optimal score for } g \text{ so far}$

7.	update the optimal pattern and optimal score for $g$
8.	$updated \leftarrow true$
9.	if (not $updated$ )
10.	continue
11.	$C \leftarrow$ all subgraph patterns with one more edge attached to $p$
12.	for each pattern $q$ in $C$
13.	if $q$ has not been generated before
14.	put $q$ into $candidate\_list$
15.	return the optimal pattern for each $g$ in $G_p$

I defined an indicator function for a subgraph pattern  $p$  as follows:

$$d(p) = \begin{cases} 1, \exists g \in G_p, score(p) > \text{optimal score for } g \text{ so far} \\ 0, \text{otherwise} \end{cases}$$

If function  $d$  is antimonotonic as patterns are extended, then when a pattern  $p$  is visited and it fails to be the optimal pattern for any positive graph (i.e.  $d(p) = 0$ ), the search process can safely prune  $p$  and any lineages extended from  $p$ . No supergraph of  $p$  will be the optimal pattern for any positive graph because of the antimonotonicity property that once  $d(p) = 0$  no supergraph  $q$  of  $p$  will have  $d(q) = 1$ . If this assumption is true, then the search process would become very efficient as only good patterns need to be considered. And single lineage exploration would have been sufficient.

However, this assumption is not always true because discrimination scores of patterns may increase as patterns become larger. Therefore, even if a pattern  $p$  is not the optimal pattern for any positive graph, a supergraph of  $p$  may be the optimal pattern for some positive graph because its score is greater than the score of  $p$ .

Nevertheless, the assumption does not have to hold for all subgraph patterns to make *fast-probe* work. In fact, for the most discriminative subgraph pattern, as long as the assumption holds for at least one of its lineages, the optimal pattern will be found. Using multiple-lineage exploration helps because the likelihood of the assumption being true for at

least one lineage is much larger in multiple-lineage exploration than in single-lineage exploration. In addition, the most discriminative subgraph pattern will not be missed as long as patterns in its lineages are optimal patterns for one positive graph at the time they are visited. This is very likely to be true: it is typical that some positive graphs are covered by multiple highly discriminative subgraphs while others do not have highly discriminative subgraphs. I call the former as “rich” graphs and the latter as “poor” graphs. Ancestors for the highly discriminative subgraphs for “rich” graphs may cover “poor” graphs when their positive frequencies are still high. Let  $p$  be the most discriminative subgraph for a “rich” graph  $g$  and  $q$  be another highly discriminative subgraph for  $g$ . Let  $q$  be visited before any ancestor of  $p$  is visited. Patterns in the lineages of  $p$  may not be the optimal patterns for  $g$  when they are visited because they may not be as discriminative as  $q$ . However they may be the optimal patterns for some “poor” graphs and thus survive and produce a lineage to  $p$ . The most discriminative subgraphs for “poor” graphs may be missed when there are no “poorer” graphs for their ancestors to survive. In this case, a subsequent (branch and bound) search may be needed to recover the most discriminative subgraphs missed by *fast-probe*.

### 5.3 Upper-bound Estimation by Learning from Search History

A tight estimated upper-bound of scores may improve the efficiency of branch-and-bound algorithms. For example, when  $p$  is visited, let the optimal score of any patterns visited so far be 1.2. If the estimated upper-bound is 1.5 (loose), then the algorithm cannot prune any descendants of  $p$ ; but if the estimated upper-bound is 1.1 (tight), then the algorithm can prune all descendants of  $p$ .

I first studied a simple way for upper-bound estimation. According to the definition, the discrimination score increases as the negative frequency decreases, and decreases as the

positive frequency decreases. A simple estimation of upper-bound for scores of descendants of  $p$  is achieved when the positive frequency remains the same as that of  $p$  and the negative frequency is zero:

$$\hat{B}(p) = \log \frac{pfreq(p)}{\varepsilon}, \text{ where } \varepsilon \text{ is a small value to replace } 0$$

This is a very loose upper-bound especially when the positive and negative frequencies of  $p$  are high. In most cases, adding edges to  $p$  causes both positive and negative frequencies to decrease. If the negative frequency decreases faster than the positive frequency, then the pattern becomes more and more discriminative; otherwise, the pattern becomes less discriminative. If the negative frequency of  $p$  is high, many edges need to be added to it to achieve zero negative frequency and as a result the positive frequency drops significantly as well. For example, in chemical compound graphs,  $C-C$  has positive and negative frequencies almost equal to 100% as it is prevalent in chemical compounds. However, its most discriminative descendants typically have positive frequency less than 15% and negative frequency close to zero. Therefore, the optimal discrimination score is much lower than the estimated upper-bound, which results in inefficient pruning.

Previous discriminative subgraph mining algorithms takes advantage of the correlations between score (or frequency) of a pattern and the largest score that the descendants of the pattern may have in designing exploration orders, in order to approach the optimal score as fast as possible. However, such correlations are qualitative and can only serve as a heuristic guidance. I proposed to learn quantitative correlations from search history and use them to estimate tight upper-bounds.

**DEFINITION 6.2 (score record).** Given a lineage of pattern  $p$ ,  $l(p) = p_1p_2...p_{k-1}p_k$ , the score record for  $l(p)$  is a sequence of scores for the patterns in the lineage:

$$h(p) = \text{score}(p_1), \text{score}(p_2), \dots, \text{score}(p_{k-1}), \text{score}(p_k)$$

A discriminative subgraph mining process always generates many score records, which can be organized into a prefix tree, called *prediction tree*. Figure 5.3 shows an example of score records and the corresponding *prediction tree*. Each tree node is labeled with score and the root node is labeled with 0.0, which is the score of an empty subgraph. In my implementation, I discretized scores evenly into 10 bins and used the discretized scores as labels. In the example, I used the original scores as labels for the sake of intuitive illustration. In addition to the score label, each tree node is also associated with the maximum score in the sub-tree rooted at this node. The score records and the corresponding *prediction tree* can be considered as a sample of the whole search space. Therefore, the maximum score at each tree node is an estimated upper-bound in the search space. For example, for a pattern  $p$  with score record (0.5, 0.7, 1.0), its maximum score in the *prediction tree* is 1.5 and thus its estimated upper-bound in the search space is 1.5. I organize the sample space by scores (rather than by subgraph structures in the search space) because it is much easier to compare scores than structures. Sometimes the score record of a pattern  $p$  is absent in the tree, so I additionally generate a lookup table, named *prediction table*, to aggregate the information in the tree. The key for each entry in the *prediction table* is composed of the number of edges in the pattern and the score of the pattern. The value stored at each entry is the maximum score of the descendants of the patterns with the corresponding size and score in the sample space. For example, if the score record of  $p$  is (0.4, 0.8), which cannot be found in the *prediction tree*, then I use the key  $\langle 2, 0.8 \rangle$  to look for an upper-bound estimation in the *prediction table*, which returns 1.0. The search history  $H$  is composed of the *prediction tree* and the *prediction table*. If neither the score record nor the  $\langle \text{size}, \text{score} \rangle$  pair of  $p$  can be found in  $H$ ,

then I use the loose upper-bound estimation discussed earlier in this section. The algorithm for upper-bound estimation based on history  $H$  is summarized as follows.

**Algorithm:** *prediction\_tree\_search*( $h(p)$ ,  $par$ ,  $cur$ )

$p$ : a subgraph pattern

$h(p)$ : the score record for a lineage of  $p$

$par$ : a tree node in the prediction tree of search history

$cur$ : an integer number indicating the current level in the prefix tree

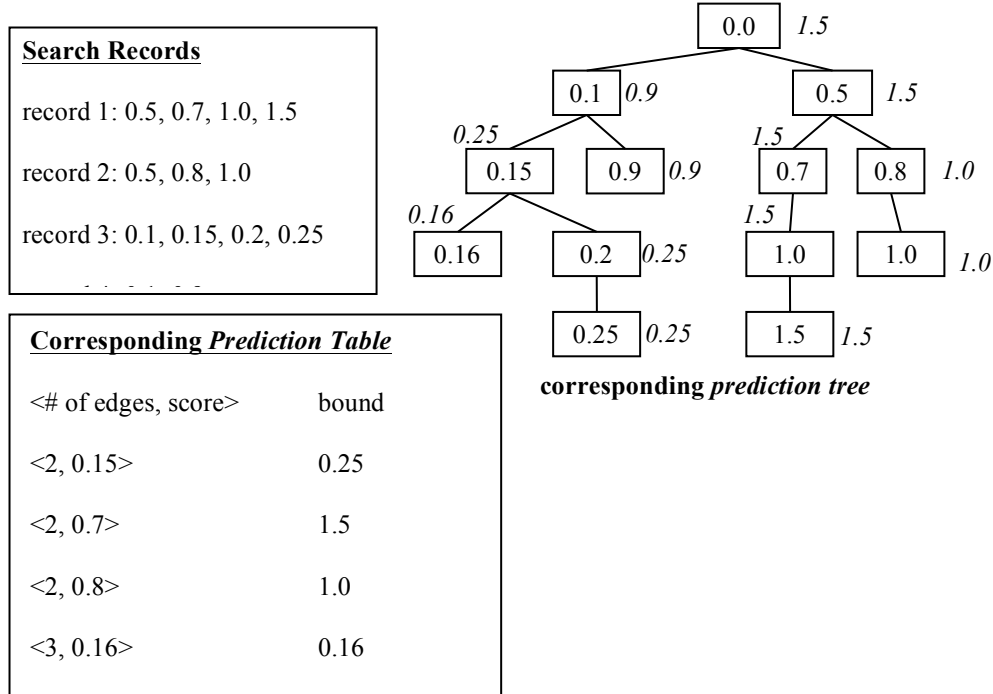
1.  $n \leftarrow$  the child node of  $par$  whose score equals  $score(p_{cur})$  in  $h(p)$
2. **if** ( $n$  is empty)
3.     **return** empty
4. **if** ( $cur = |E(p)|$ )
5.     **return**  $n$
6. **else**
7.     **return** *prediction\_tree\_search*( $h(p)$ ,  $n$ ,  $cur+1$ )

**Algorithm:** *upper\_bound\_estimation*( $p$ ,  $H$ )

$p$ : a subgraph pattern

$H$ : search history, including a prediction tree and a prediction table

1.  $h(p) = score(p_1), score(p_2), \dots, score(p_{k-1}), score(p_k)$ , where  $l(p) = p_1 p_2 \dots p_{k-1} p_k$  is the current lineage through which  $p$  is visited
2.  $n \leftarrow$  *prediction\_tree\_search*( $h(p)$ ,  $H.prefix\_tree.root$ , 1)
3. **if** ( $n$  is not empty)
4.     **return**  $n.empirical\_upper\_bound$
5. **else if** ( $H.lookup\_table[|E(p)|, score(p)].exists$ )
6.     **return**  $H.lookup\_table[|E(p)|, score(p)]$
7. **else**
8.     **return**  $\log \frac{pfreq(p)}{\epsilon}$



**Figure 5.3: An example of search records and the corresponding *prediction tree* and *prediction table***

Using search history to estimate upper-bound bears the risk of underestimating upper-bound if the discriminative subgraph mining process, which provides the score records, fails to capture a good sample of high discrimination scores. This will result in inefficient pruning and thus prolonged execution time. However, there is little impact to the mining process if the greedy sampling misses many low discrimination scores because, although these score records may be absent in the *prediction tree*, the *prediction table* can still provide a reasonably tight upper-bound estimation and the algorithm always has the last resort to the loose estimation.

LTS first uses *fast-probe* to collect score records and generates search history  $H$ , which includes a *prediction tree* of score records and a *prediction table* aggregating the score records. LTS utilizes a vector  $F$  to keep track of the optimal pattern for each positive graph:



$F[i]$  stores the optimal pattern for positive graph  $g_i$ . Vector  $F$  is updated with the optimal patterns found by *fast-probe*, which compose a better starting point than single-edge subgraphs, before the following branch-and-bound search. Then LTS performs a branch-and-bound search in the subgraph search space and uses a candidate list to keep track of candidate subgraph patterns. Its goal is to find the most discriminative subgraph for each positive graph. When the branch-and-bound search begins, the candidate list is initialized with all subgraphs with one edge. LTS repeatedly pops one subgraph from the candidate list at a time until the candidate list becomes empty. LTS uses CCAM code to encode subgraphs and maintains a lookup table to keep track of processed subgraphs. For each subgraph  $p$  from the candidate list, LTS updates  $F[i]$  if positive graph  $g_i$  supports  $p$  and  $score(p)$  is greater than  $score(F[i])$ . Meanwhile, LTS estimates the upper-bound of  $p$  based on search history  $H$  and checks whether the upper-bound is greater than any  $score(F[i])$  with  $g_i$  supporting  $p$ . If the upper-bound is not greater than the optimal score of any positive graph supporting  $p$ , then  $p$  is discarded from further extension. Note that for each pattern, the algorithm only considers the positive graphs supporting this pattern when updating optimal scores and pruning with the estimated upper-bound because the algorithm is looking for the optimal pattern for each positive graph. If  $p$  is preserved, LTS computes all of its extensions with one more edge in the positive set. The extensions that have not been visited before are put into the candidate list.

The algorithm of LTS is summarized as follows.

**Algorithm:**  $LTS(G_p, G_n)$

$G_p$ : positive graph set

$G_n$ : negative graph set

$F$ : a vector maintaining the optimal pattern for each positive graph

*Candidate\_list*: the set of subgraph patterns to be extended

$H$ : search history of the patterns visited by *fast-probe*

```

1.  $F \leftarrow \text{fast-probe}(G_p, G_n)$  and store search history of the patterns visited by fast-probe in  $H$ 
2. Put all single-edge subgraph pattern into candidate_list
3. while (candidate_list is not empty)
4.    $p \leftarrow$  get next pattern and remove it from candidate_list
5.    $\text{is\_promising} \leftarrow \text{false}$ 
6.   for each graph  $g$  in  $G_p$  that supports  $p$ 
7.     if  $\text{upper\_bound\_estimation}(p, H) > \text{score}(F[i])$ 
8.        $\text{is\_promising} \leftarrow \text{true}$ 
9.       if  $\text{score}(p) > \text{score}(F[i])$ 
10.         $F[i] \leftarrow p$ 
11.   if (not  $\text{is\_promising}$ )
12.     continue
13.    $C \leftarrow$  all subgraph patterns with one more edge attached to  $p$ 
14.   for each pattern  $q$  in  $C$ 
15.     if  $q$  has not been generated before
16.       put  $q$  into candidate_list
17. return the optimal pattern for each  $g$  in  $G_p$ 

```

## 5.4 Experiments

In the experiments, I used discriminative subgraphs found by *fast-probe* or LTS to perform graph classification and measure the performance of *fast-probe* or LTS by its runtime and classification accuracy. In the end of this section, I also compared the best score found by LTS and the best score found by LEAP.

### 5.4.1 Power of Multiple-lineage Exploration

This section studies the importance of using multiple-lineage exploration rather than single-lineage exploration. I implemented two versions of *fast-probe*: one with multiple-lineage exploration and the other with single-lineage exploration, respectively. I refer to the former as *ME-fast-probe* and the latter as *SE-fast-probe*. Figure 5.4 shows the normalized accuracy comparison between the two versions of *fast-probe* for chemical compound datasets. *SE-fast-probe* is obviously incompetent in finding discriminative subgraphs for its accuracy is around 50%, while *ME-fast-probe* has much better performance with an average accuracy around 70%. The low accuracy of *SE-fast-probe* is due to its intolerance of missing

patterns and the aggressive pruning strategy of *fast-probe*. *ME-fast-probe* benefits from its redundancy in exploration and is able to find discriminative subgraph patterns in spite of the extremely aggressive pruning strategy. Figure 5.5 compares the runtime of the two versions of *fast-probe*. Although *ME-fast-probe* is slower than *SE-fast-probe*, the difference is acceptable. The redundancy in exploration of *ME-fast-probe* does not lead to very poor runtime performance because many subgraphs are removed from extension by the pruning strategy. Therefore, the redundant exploration method and the highly aggressive pruning strategy complement each other well.

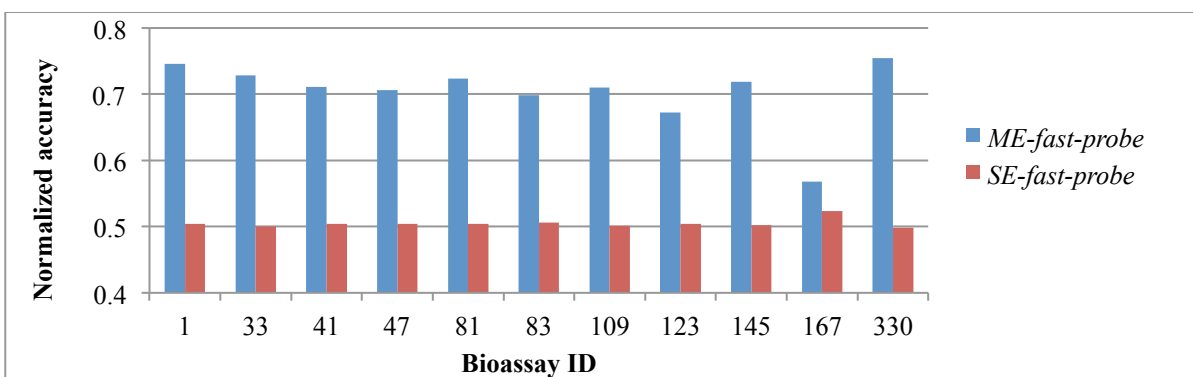


Figure 5.4: Normalized accuracy comparison between multiple-lineage-exploration-based *fast-probe* and single-lineage-exploration-based *fast-probe* using chemical datasets

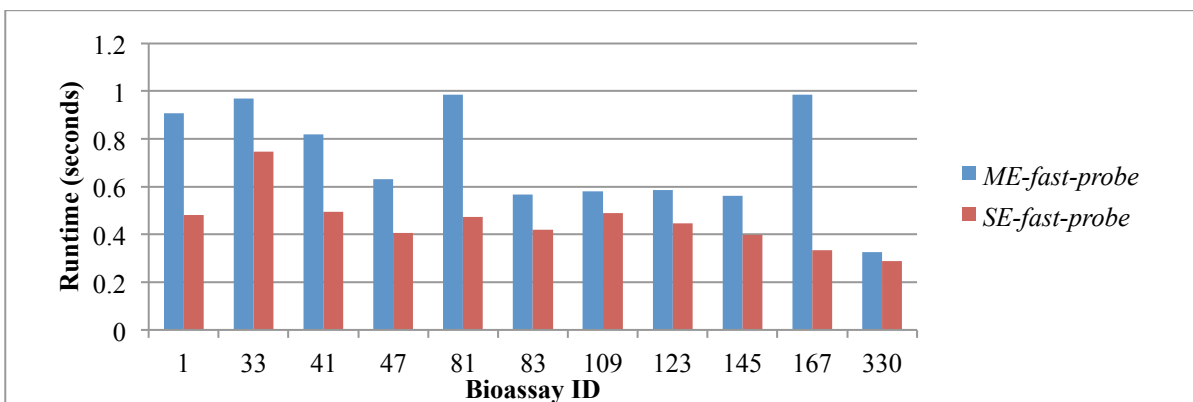


Figure 5.5: Runtime comparison between multiple-lineage-exploration-based *fast-probe* and single-lineage-exploration-based *fast-probe* using chemical datasets

### 5.4.2 Efficiency of *Fast-probe*

I also compared *fast-probe* with three state-of-the-art discriminative subgraph mining algorithms, GAIA [Jin2010], COM [Jin2009] and graphSig [Ranu2009], using the chemical compound datasets. The parameter settings (listed in Table 5.1) for GAIA, COM and graphSig are the same as those used in [Jin2010], [Jin2009] and [Ranu2009] for chemical datasets since the same datasets are used here. The number of CPUs used by GAIA is set to be 1 in order to optimize its normalized accuracy. Using more than one CPU for chemical datasets does not lead to higher accuracy for GAIA. There is no parameter for *fast-probe*.

**Table 5.1: Parameter settings for GAIA, COM and graphSig for chemical datasets**

Algorithm	Parameters for chemical datasets
GAIA	Candidate list size = 10, maximal # of iterations = 4, # of CPUs used = 1
COM	Positive frequency threshold = 1%, Negative frequency threshold = 0.4% Maximal # of edges in a subgraph = 5
graphSig	maxPvalue=0.1, minFreq=0.1%

Figure 5.6 compares the normalized accuracy between the four algorithms. On average, the normalized accuracy of *fast-probe* is 6.07% higher than that of COM and 2.67% higher than that of graphSig. I performed paired t-test to evaluate the statistical significance of such difference in normalized accuracy. The lower the p-value is, the more statistically significant the difference is. The p-value for the improvement over graphSig is 0.003 and the p-value for the improvement over COM is  $1.72 \times 10^{-8}$ . GAIA and *fast-probe* have similar normalized accuracy (GAIA's accuracy slightly lower by 0.06%). The difference is not statistically significant, but *fast-probe* runs faster than GAIA. Figure 5.7 shows the runtime comparison between the four algorithms. On average, *fast-probe* is 4.76 times faster than

COM and 34.35 times faster than graphSig. *Fast-probe* outruns GAIA in 9 out of 11 chemical compound datasets and its average speed is 1.8 times faster. The differences in runtimes are statistically significant at the 0.005 level.

This experiment shows that *fast-probe* alone is a highly competitive discriminative subgraph mining method for chemical compound datasets.

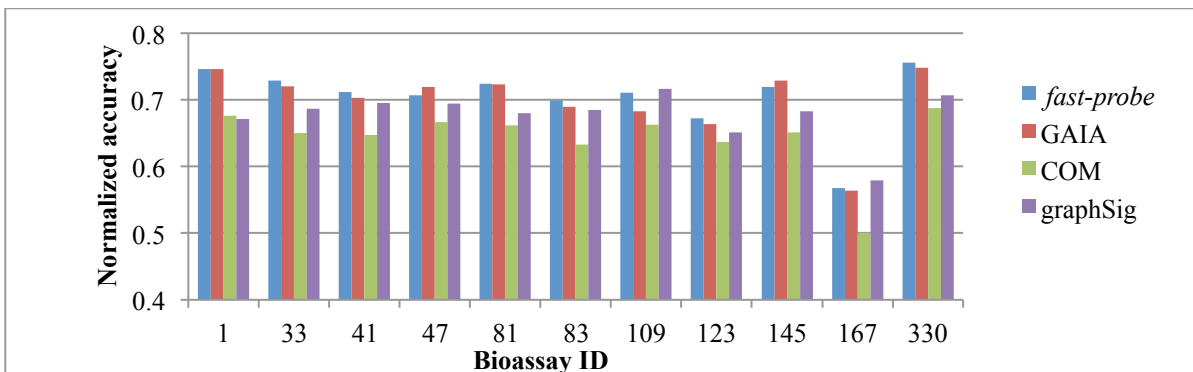


Figure 5.6: Normalized accuracy comparison between *fast-probe*, GAIA, COM and graphSig using chemical datasets

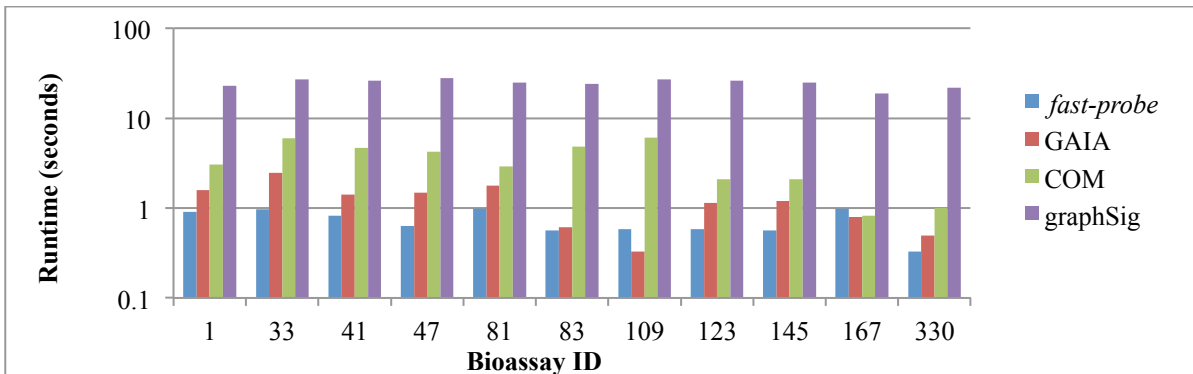
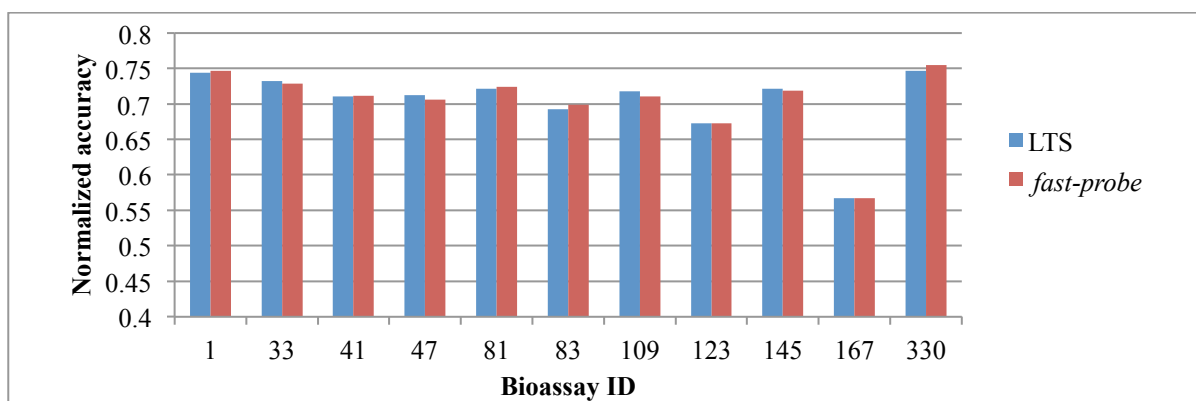


Figure 5.7: Runtime comparison between *fast-probe*, GAIA, COM and graphSig using chemical datasets

### 5.4.3 Effectiveness of Using Search History

This section compares the performance of *fast-probe* and LTS. *Fast-probe* is part of LTS and is invoked first to generate search history. Figure 5.8 shows the normalized accuracy comparison between *fast-probe* and LTS for chemical datasets. It can be seen that

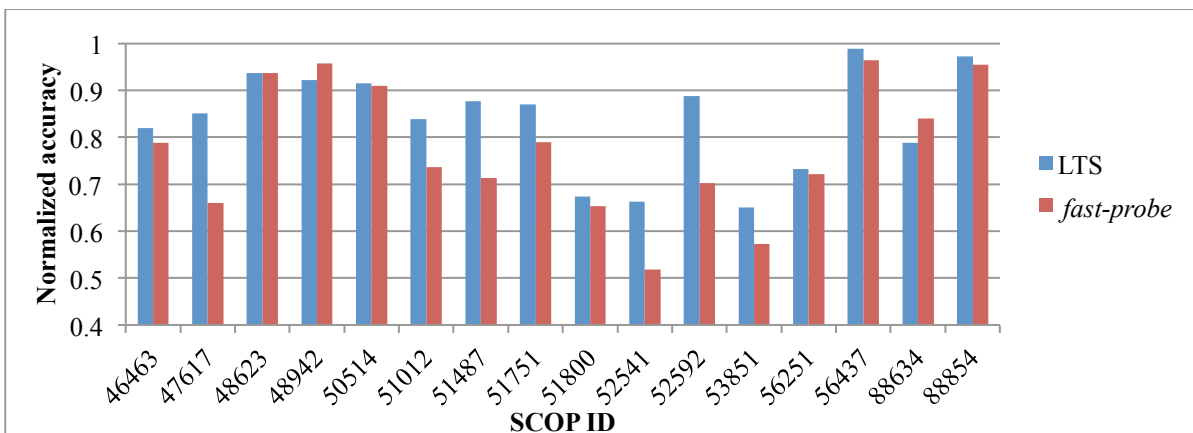
LTS and *fast-probe* produce almost same classification accuracy in chemical compound datasets. LTS cannot further improve the classification accuracy of *fast-probe* in chemical datasets because *fast-probe* is already very efficient for these datasets and has comparable, if not better, performance compared with state-of-the-art algorithms. When the highly discriminative subgraph patterns are already found, further search in the pattern space cannot lead to better classification accuracy unless more sophisticated classifier generation methods are used. LTS has slightly worse result than *fast-probe* for some chemical datasets because higher discrimination scores in the training set do not necessarily guarantee higher classification accuracy in the test set when the difference in discrimination scores is marginal. Discrimination score is calculated based on the training set while the classification is performed on the test set. If the score of  $p$  is only slightly greater than the score of  $q$  in the training set, it is possible that the score of  $p$  is slightly less than that of  $q$  in the test set. Selecting the pattern with higher score in the training set may sometimes produce slightly lower accuracy in the test set.



**Figure 5.8: Normalized accuracy comparison between *fast-probe* alone and LTS using chemical datasets**

Figure 5.9 compares the normalized accuracy of LTS and *fast-probe* using protein datasets. LTS outperforms *fast-probe* in terms of accuracy in 14 out of 16 protein datasets. In

5 protein datasets LTS improves accuracy by more than 10%. On average, LTS improve normalized accuracy by 8.06% compared with *fast-probe* and the p-value for this improvement is 0.003.



**Figure 5.9: Normalized accuracy comparison between *fast-probe* alone and LTS using protein datasets**

The results of the two comparisons in chemical datasets and protein datasets are consistent with the fact that protein graphs are much more complex than chemical graphs for protein graphs have more nodes, higher edge degrees and more diverse labels (chemical graphs have more labels, but the majority of nodes are labeled as Carbon, Oxygen or Nitrogen and most edge are labeled as single-bond). It is harder to mine the most discriminative subgraph patterns in protein graphs because of the much larger search space. As a result, *fast-probe* is relatively incompetent and LTS is able to improve accuracy significantly for protein graphs. This is contrary to the case of chemical graphs where *fast-probe* is highly efficient and LTS does not improve its accuracy.

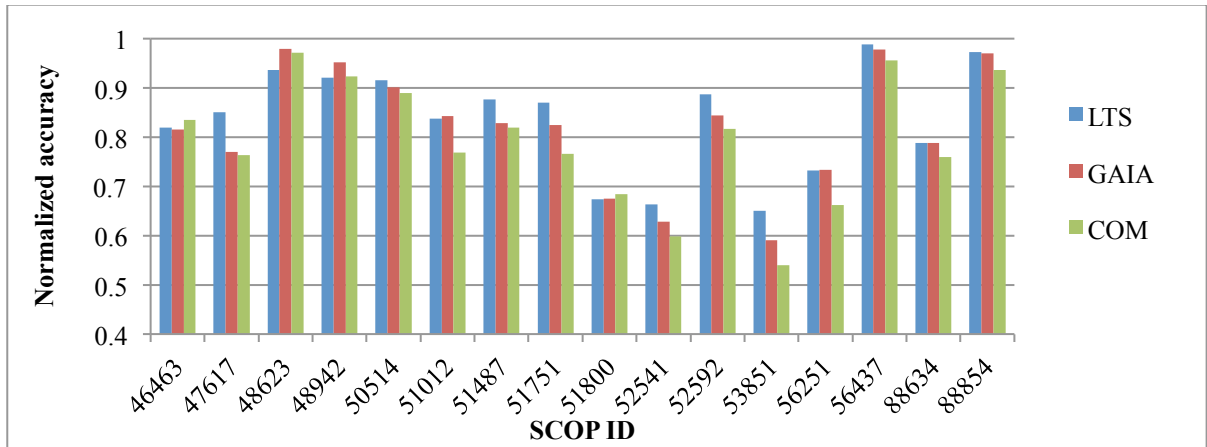
For protein graphs, I compared LTS with GAIA and COM (graphSig is optimized for chemical graphs, so it achieves only ~50% normalized accuracy for protein graphs and takes significantly longer time than COM and GAIA). The parameter settings (listed in Table 5.2) are the same as those used in [Jin2009] and [Jin2010] for protein graphs, respectively. The

number of CPUs is set to be 32 for GAIA to optimize the accuracy. When the number of CPUs used by GAIA is less than 8, the classification accuracy of GAIA is lower than that of COM. Note that LTS is parameter free.

**Table 5.2: Parameter settings for GAIA and COM for protein datasets**

Algorithm	Parameters for protein datasets
GAIA	Candidate list size = 100, maximal # of iterations = 10, # of CPUs used = 32
COM	Positive frequency threshold = 30%, Negative frequency threshold = 0.0%

Figure 5.10 shows the normalized accuracy comparison between LTS, GAIA and COM using protein datasets. LTS outperforms GAIA in 11 out of 16 protein datasets and excels COM in 12 out of 16 datasets in terms of normalized accuracy. The average normalized accuracy of LTS is 1.63% higher than that of GAIA and 4.32% higher than that of COM. The p-value for LTS's improvement over GAIA is 0.033 and the p-value for LTS's improvement over COM is 0.0006.

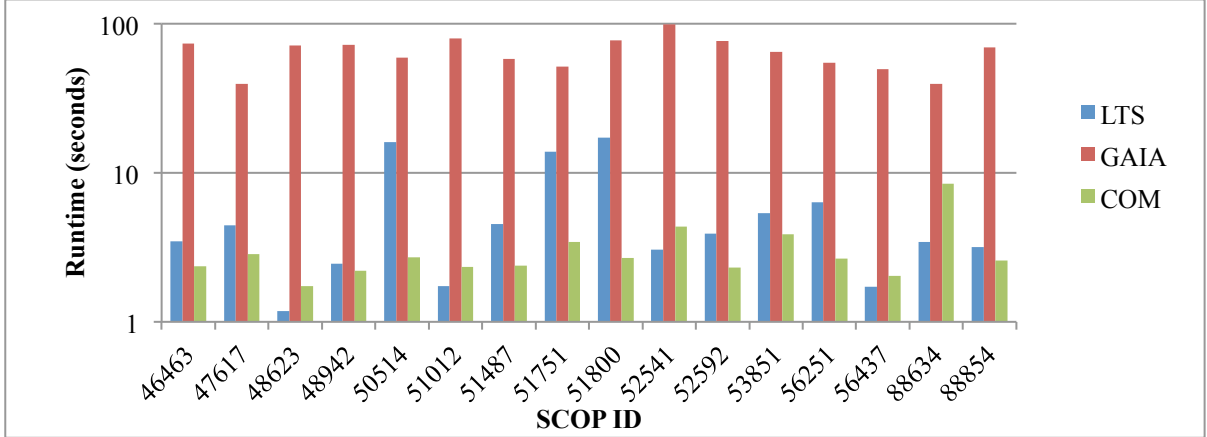


**Figure 5.10: Normalized accuracy comparison between LTS, GAIA and COM using protein datasets**

Figure 5.11 compares the runtime of LTS, GAIA and COM. In order to reflect the search efficiency, I multiplied the runtime of GAIA by 32 for it uses 32 processes searching



in parallel. Even though GAIA returns the result in a short amount of time  $t$ , its actual search time is much longer than what  $t$  indicates. It can be seen that after taking the total CPU cycles consumed as runtime, GAIA becomes an order of magnitude slower than LTS and COM. In most protein datasets, LTS has comparable speed with COM. On average, LTS is 11.3 times faster than GAIA when the actual computation time is considered and 1.88 times slower than COM with significantly higher accuracy. In spite of this, the runtime efficiency of LTS is still remarkable. I also implemented a naive branch-and-bound search using the simplest upper-bound estimation (*fast-probe* is used to provide starting optimal scores). It runs for more than 10 minutes before it runs out of memory in the end on a protein dataset. Besides, COM requires user input in setting the suitable positive and negative frequency thresholds which determines the efficiency of the search space pruning. LTS does not require such information from users.



**Figure 5.11: Runtime comparison between LTS, GAIA and COM using protein datasets**

#### 5.4.4 Quality of Individual Patterns

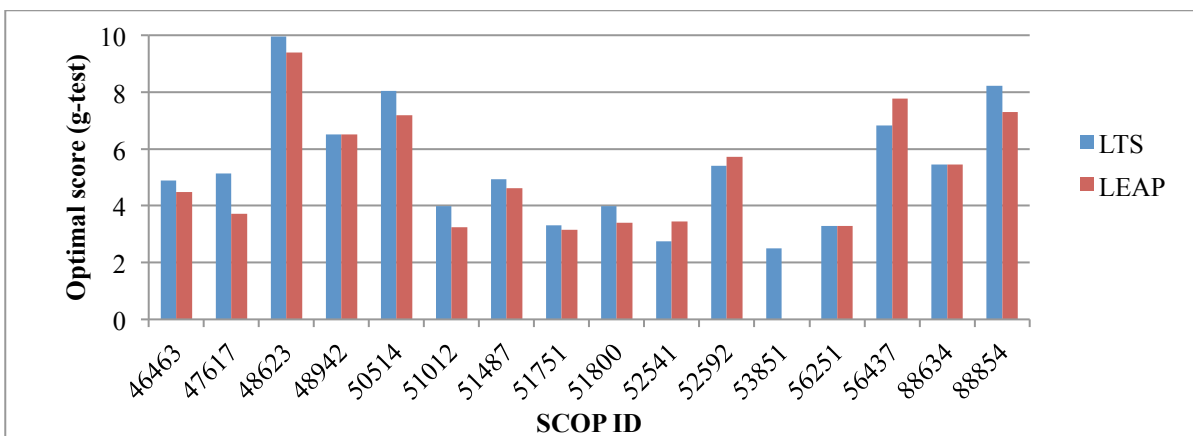
Although LTS aims to find a set of discriminative subgraph patterns rather than the most discriminative subgraph pattern as LEAP does, the high classification accuracy of LTS is not a mere result of complementary weak features. In fact, the best subgraph pattern found

by LTS is as good as, if not better than, the best pattern found by LEAP. I ran both LTS and LEAP to look for the subgraph pattern with the highest g-test score for each dataset (I used g-test in this experiment because it is used in the original implementation of LEAP). The g-test score of a pattern  $p$  is defined as:

$$pfreq(p) * \log \frac{pfreq(p)}{nfreq(p)} + (1 - pfreq(p)) * \log \frac{1 - pfreq(p)}{1 - nfreq(p)}$$

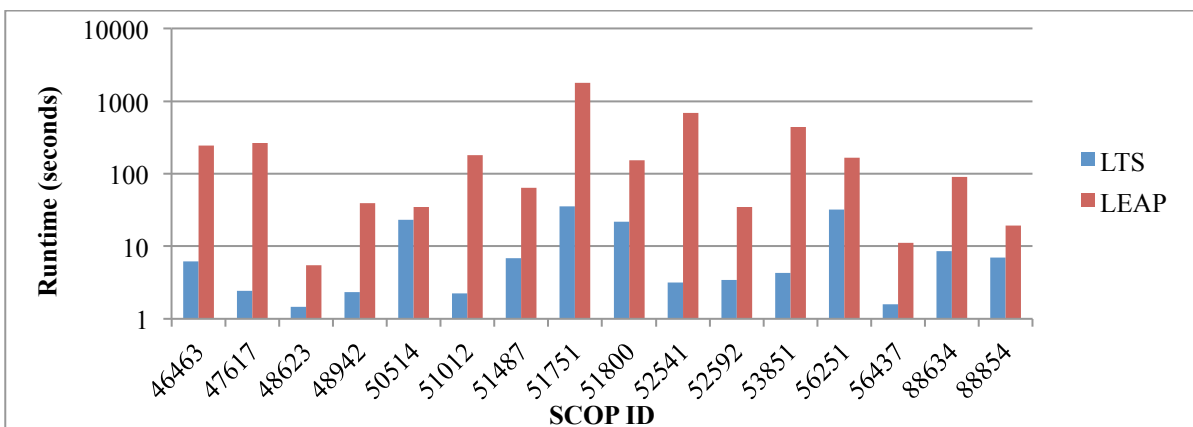
LEAP has a parameter  $\sigma$  to control the leap length. The larger the value of  $\sigma$ , the faster the search. But a large  $\sigma$  often results in low optimal score found by LEAP. I set the leap length  $\sigma$  to be 0.05 which is the lowest leap length LEAP can handle without running out of memory. I also set the terminating positive frequency threshold to be 20% for LEAP's frequency-descending mining for protein datasets because all discriminative subgraph patterns found by LTS, GAIA and COM in protein datasets have positive frequency greater than 20%. For chemical compound datasets, I set the terminating positive frequency threshold to be 5% for LEAP's frequency-descending mining. LTS and *fast-probe* are parameter free.

Figure 5.12 compares the optimal scores found by LTS and LEAP in protein datasets. Out of the 16 protein datasets, LTS has higher optimal score than LEAP in 10 datasets and has same optimal score with LEAP in 3 datasets. LEAP has better optimal score than LTS in only 3 out of 16 protein datasets. On average, the optimal score found by LTS is higher than that of LEAP by 0.4 and the p-value for this improvement is 0.03.



**Figure 5.12: Optimal score comparison between LTS and LEAP using protein datasets**

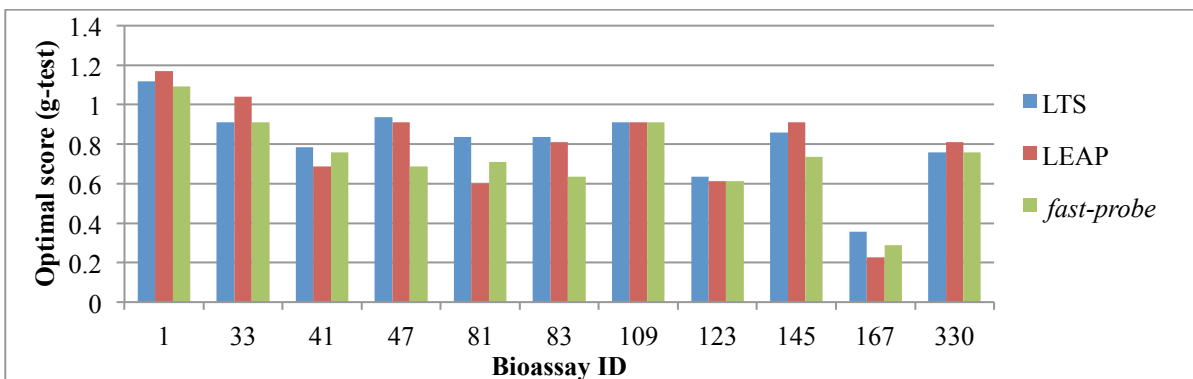
Figure 5.13 shows the runtime comparison between LTS and LEAP using protein datasets. LTS always outruns LEAP and the average speed of LTS is 50 times faster than that of LEAP. The p-value for this improvement in runtime is 0.018.



**Figure 5.13: Runtime comparison between LTS and LEAP using protein datasets**

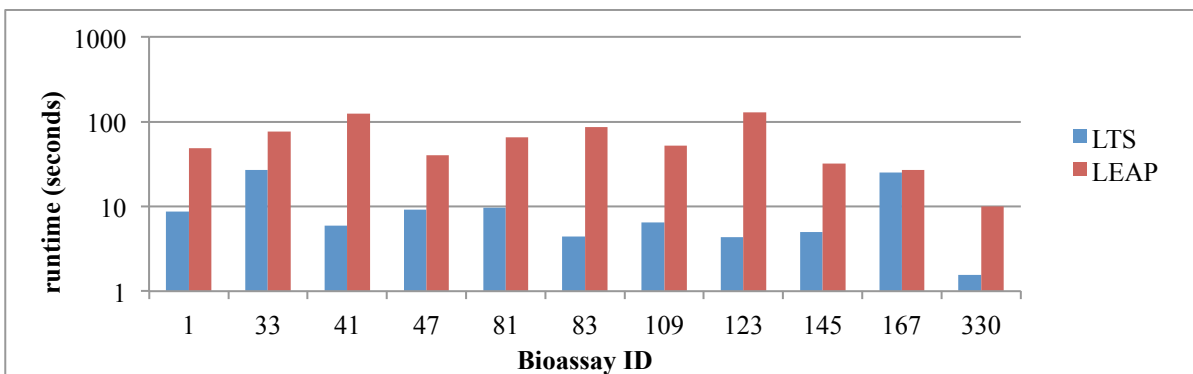
Figure 5.14 compares the optimal scores found by LTS, LEAP and *fast-probe* in chemical datasets. Although *fast-probe* has the same classification accuracy as LTS (shown in Figure 5.8), the optimal patterns found by *fast-probe* are not as good as those found by LTS. On average, the optimal scores found by LTS are higher than those from *fast-probe* by 0.076. The p-value for this difference is 0.0077. When comparing LTS and LEAP using chemical datasets, it can be seen that they have very similar performance in terms of optimal

scores. Out of 11 chemical datasets, LTS has higher optimal scores than LEAP in 6 datasets, lower optimal scores in 4 datasets and same score in 1 dataset. The average optimal score of LTS is slightly higher than that of LEAP by 0.023. The p-value for this difference is 0.46, which is statistically insignificant.



**Figure 5.14: Optimal score comparison between LTS, LEAP and *fast-probe* using chemical compound datasets**

Figure 5.15 shows the runtime comparison between LTS and LEAP using chemical datasets. LTS is always faster than LEAP. The average speed of LTS is 6.5 times faster than that of LEAP. The p-value for this runtime improvement is 0.0007.



**Figure 5.15: Runtime comparison between LTS and LEAP using chemical compound datasets**

## Chapter 6

### Mining Discriminative GG-subgraph Patterns

Existing discriminative substructure pattern mining algorithms search for rigid subgraph patterns that separate one class of graphs from another. Flexibility in subgraph patterns is not permitted. As a result, existing algorithms are unable to find highly discriminative substructure patterns with varying appearances in the dataset, due to the presence of noise and/or lack of structural rigidity in the discriminative substructure patterns. One potential solution to this problem is approximate subgraph mining, which tolerates mismatches between a subgraph pattern and a graph when calculating subgraph frequency. Zhang et al. propose two approximate frequent subgraph mining algorithms Monkey [Zhang2007] and RAM [Zhang2008] to tolerate mismatches between a pattern and a graph as long as the edit distance is within a user-specified threshold. One major drawback of these two algorithms is the explosion in the number of approximate frequent subgraph patterns as a result of approximation. In addition, in [Zhang2007] and [Zhang2008], approximate subgraph patterns are usually larger than typical rigid subgraph patterns, and therefore require more operations to compute. Besides, they allow any possible mismatches as long as the edit distance is within a user-specified threshold. It is not an efficient candidate model for mining flexible discriminative substructure patterns because not every allowed mismatch is useful in improving discrimination power. Another related model is proximity pattern mining [Khan2010], which transforms the subgraph mining problem to a probabilistic itemset

mining problem by propagation of node labels. However, the model allows so much relaxation that the preserved structural information is minimal. Therefore it is only suitable for large sparse network graphs in which closeness between nodes is more important than the topology of the connections.

Another issue of existing discriminative substructure (subgraph) pattern mining algorithms is that they require many expensive edge extension operations. Existing discriminative substructure mining algorithms start with all possible one-edge substructures and extend small substructures into large ones by adding one edge at a time. As a result, generating a substructure pattern with  $n$  edges requires  $n-1$  edge extensions. Edge extensions are time consuming due to the large number of possible extensions. One way to make discriminative substructure pattern mining more efficient is to reduce the number of edge extensions required to generate a pattern.

To capture flexible discriminative substructure patterns, I proposed a new type of substructure patterns, GG (Graph on Graphs)-subgraph patterns. GG-subgraph patterns are a generalization of conventional subgraph patterns. A conventional subgraph pattern is a graph with atomic nodes and edges. A GG-subgraph pattern also has “nodes” and “edges” (denoted as GG-nodes and GG-edges respectively): each GG-node is itself a graph and each GG-edge connects a pair of GG-nodes. Each GG-node in a GG-subgraph pattern corresponds to a rigid subgraph pattern in an original graph. Two GG-nodes are connected by a GG-edge in a GG-subgraph pattern if the two corresponding subgraph patterns are incident or overlapping in the original graph. GG-edges in GG-subgraph patterns are not labeled. Thus, how two GG-nodes are structurally connected is only loosely described in a GG-subgraph pattern. The Rigidity in GG-nodes and flexibility in GG-edges enable GG-subgraph patterns to capture

flexible substructures at a higher level than previous approximate subgraph patterns without explosion in the number of patterns. As a result, a GG-subgraph pattern can be considered as a substructure pattern consisting of small rigid component patterns assembled in a flexible way. On the contrary, a conventional subgraph pattern is a special case of a GG-subgraph pattern having only one GG-node, without any flexibility.

To reduce the number of edge extensions needed to generate patterns, I proposed an algorithm, GG-miner, to mine discriminative GG-subgraph patterns. It first uses conventional discriminative subgraph mining to find rigid discriminative subgraph patterns. This first step is based on an observation that rigid subgraph patterns with high discrimination power are much more likely than patterns with high frequency to be present in larger discriminative patterns. Then in the second step, given the rigid patterns, GG-miner transforms the original graphs to GG graphs, in which each GG-node is itself a rigid pattern. GG-miner then uses conventional discriminative subgraph mining again to search GG graphs for GG-subgraph patterns. In the mining process, the transformation step not only provides candidate GG-nodes for the subsequent computation of discriminative GG-subgraph patterns but also reduces the number of necessary edge extensions. Fewer edge extensions are needed because each GG-node represents multiple nodes in the original graphs and one edge extension in the GG graphs corresponds to multiple edge extensions in the original graphs.

## **6.1 Mining Discriminative GG-subgraph Patterns**

### **6.1.1 Definition of GG-subgraph Patterns**

Each GG-subgraph pattern is a GG graph. I defined GG graphs as a generalization of conventional graphs to allow structural flexibility.

**DEFINITION 7.1 (GG Graph).** A GG (Graph on Graphs) graph is denoted as  $h = (C, D)$  where  $C$  is a set of graphs and  $D$  represents connectivity of graphs in  $C$ .

**DEFINITION 7.2 (GG Transform (GT)).** Given a graph  $g$  and a graph set  $P$ , a GG transform  $GT(g, P)$  is a function that maps graph  $g$  to a GG graph  $h = (C, D)$ :

- $C = \{c \mid \exists p \in P: c \text{ is an embedding of } p \text{ in } g\}$
- $D = \{(a, b) \mid a, b \in C, a \neq b: a.V \cap b.V \neq \emptyset \text{ OR } \exists (u \in a.V, v \in b.V): (u, v) \in g.E\}$

The intuition is that every embedding of each graph in  $P$  is a GG-node in  $GT(g, P)$ . If two embeddings  $a$  and  $b$  are overlapping or incident in  $g$ , then there exists a GG-edge connecting them in  $GT(g, P)$ . Thus, how the GG-nodes are connected in the original graph is only approximately described in the transformed graph, which allows structural flexibility. Figure 6.1 shows an example of GG transformation. There are two graphs in  $P$  and each has one embedding in  $g$ . Each embedding corresponds to one GG-node in  $GT(g, P)$ :  $p$  corresponds to the black GG-node and  $q$  corresponds to the red GG-node in the GG graph. The two GG-nodes are connected in the transformed graph because the two corresponding embeddings are incident in the original graph  $g$ .



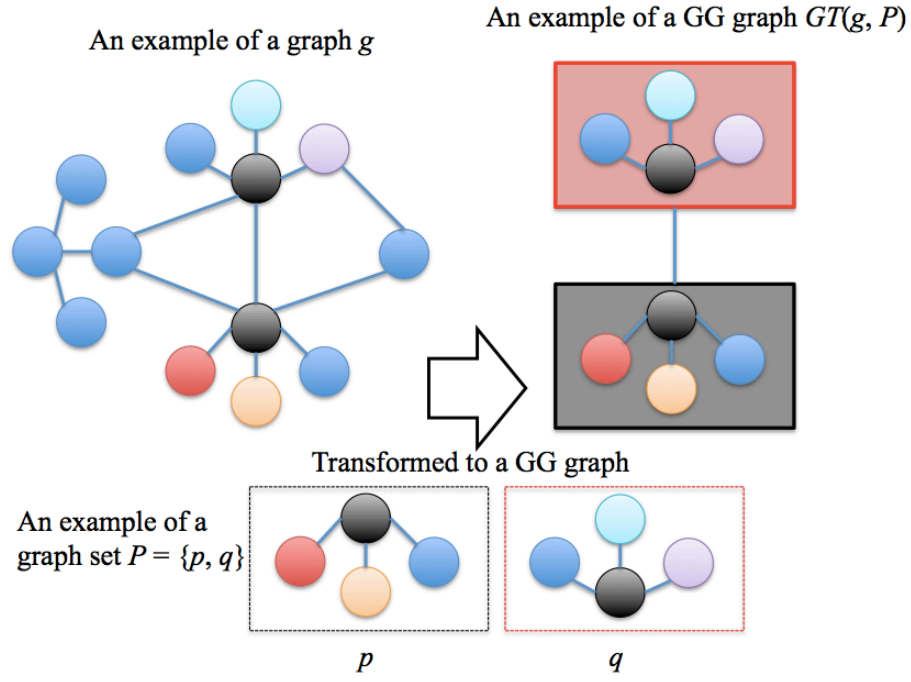


Figure 6.1: An example of GG Transformation

**DEFINITION 7.3 (Subgraph Isomorphism for GG Graphs).** Given two GG graphs  $h$  and  $h'$ , if there exists an injection  $t: h.C \rightarrow h'.C$  such that  $\forall g \in h.C: g = t(g)$  and  $\forall (g_i, g_j) \in h.D: (t(g_i), t(g_j)) \in h'.D$ , then  $h$  is a GG-subgraph of  $h'$  ( $h \subseteq h'$ ) and  $h'$  is a GG-supergraph of  $h$ .

When I need to determine whether a graph  $g$  contains a GG-subgraph pattern  $h$ , I first use  $GT(g, h.C)$  to transform  $g$  to a GG graph. What the transformation does is to find in  $g$  all the embeddings of the graphs in  $h.C$  and generate a GG graph to describe approximately how the embeddings are connected in  $g$ . After the transformation, I check whether  $h$  is a GG-subgraph of the transformed graph  $GT(g, h.C)$ . If it is the case, it means  $g$  contains all the graphs in  $h.C$  and they are connected as described in  $h.D$ . Then I consider  $g$  contains pattern  $h$ .

**DEFINITION 7.4 (Frequency of a GG-subgraph pattern).** Given a graph set  $G$  and a GG-subgraph pattern  $h$ , the frequency of  $h$  is:

$$freq(h) = \frac{|\{g \mid g \in G, h \subseteq GT(g, h.C)\}|}{|G|}$$

I denote the frequency of pattern  $h$  in the positive set by  $pfreq(h)$  and the frequency in the negative set by  $nfreq(h)$ . Then I can use the same discrimination scoring function for subgraph patterns to measure the discrimination power of GG-subgraph patterns.

### 6.1.2 Overview of Mining Discriminative GG-subgraph Patterns

Since GG-subgraph patterns are generalized graphs with flexibility in how small components are connected, any conventional discriminative subgraph mining algorithms can be used to find discriminative GG-subgraph patterns. One may consider a GG-subgraph pattern as a subgraph pattern with two types of edges: conventional rigid edges connecting nodes and special flexible edges connecting smaller subgraph patterns. However, there are three problems in this straightforward solution. First of all, it does not improve the efficiency of the mining process because it requires the same number of edge extension operations. Secondly, extension operations on flexible edges are much more time consuming than that on rigid edges. A flexible edge simply indicates that the two smaller subgraph patterns are either incident or overlapping (without specifying details). As a result, a flexible edge extension may represent many possible extensions to rigid subgraphs and thus poses challenges to subgraph isomorphism test. Thirdly, conventional subgraph mining algorithms do not support overlapping nodes. Two GG nodes may correspond to two embeddings that share some nodes in an original graph.

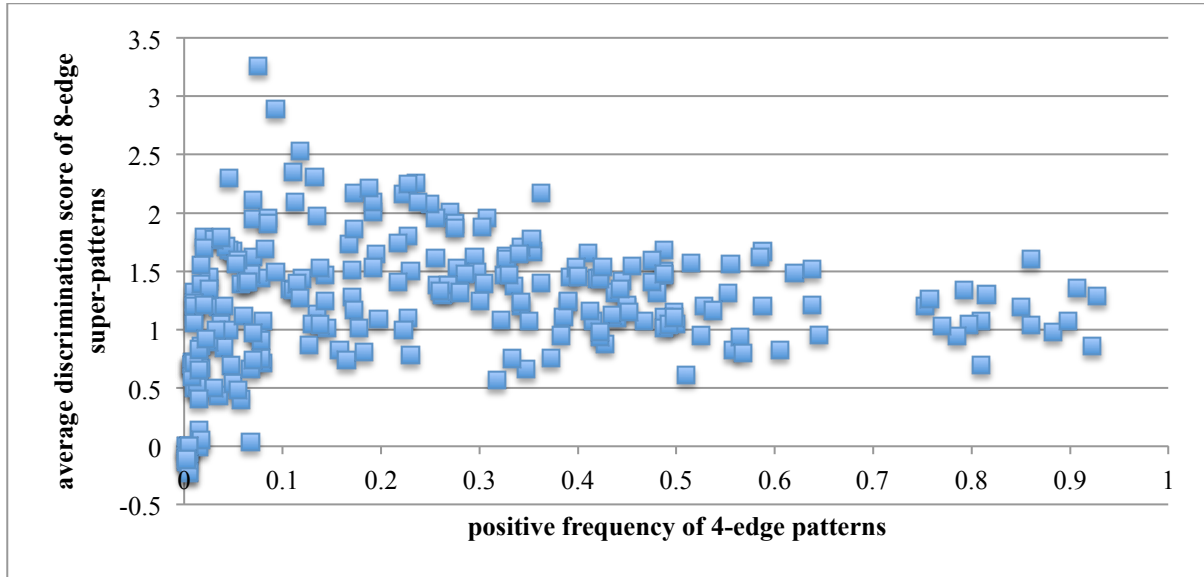
Therefore I proposed a new mining algorithm, GG-miner, to mine discriminative GG-subgraph patterns. GG-miner has two phases. In the first phase, GG-miner chooses a set of rigid subgraph patterns and uses them as candidate GG-nodes to transform original graphs to GG graphs. In the second phase, GG-miner mines discriminative GG-subgraph patterns from the GG graphs. In the end, GG-miner reports the patterns as resulting discriminative GG-subgraph patterns.

GG-miner provides a solution to the above three problems: 1) First of all, it reduces the number of edge extension operations needed to generate a pattern because each edge extension in GG graphs is paramount to multiple edge extensions in the original graphs. 2) Secondly, edge extension in GG graphs is very efficient when the candidate GG-node set for transformation is small. A small candidate set leads to small GG graphs, which have a smaller search space than the original graphs. In addition, edge extensions in GG graphs are rigid extensions. On the contrary, if I want to mine GG graph edges as a special type of edges in conventional graphs, I need to perform flexible edge extensions, which may entail a huge search space because more possibilities need to be considered. Therefore, edge extensions in GG graphs are much cheaper operations. 3) Thirdly, GG-miner supports overlapping nodes because overlapping embeddings are represented by separate GG-nodes during transformation.

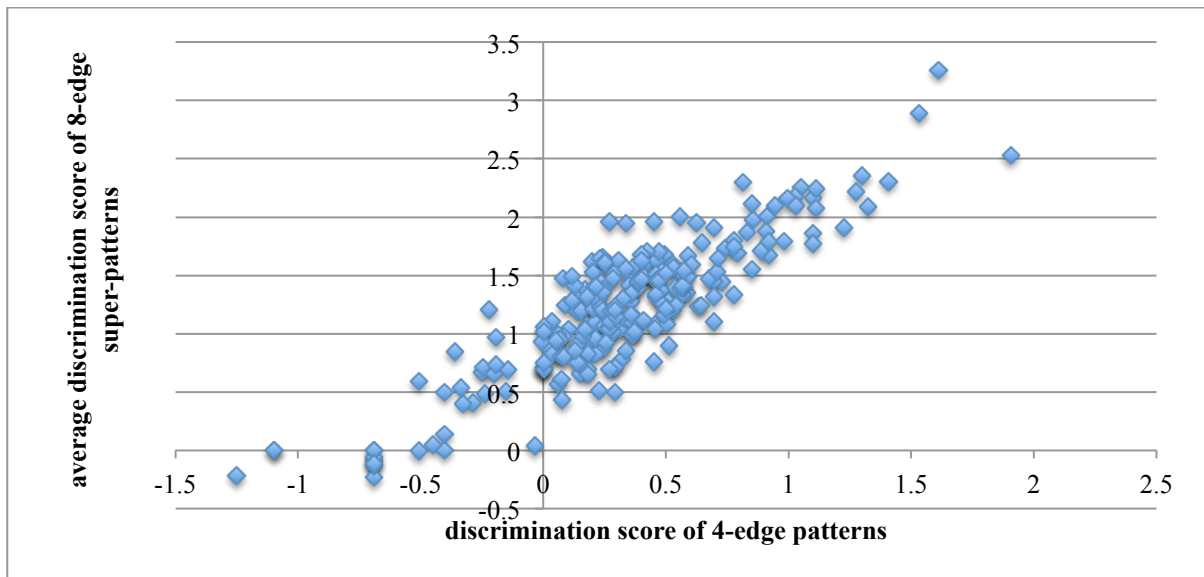
### **6.1.3 How to Choose Rigid Subgraph Patterns for Transformation**

How GG-miner choose the set of rigid subgraph patterns as candidate GG-nodes for transformation is crucial to the effectiveness and efficiency of GG-miner. Using randomly sampled subgraph patterns often misses discriminative substructure patterns. Using all subgraph patterns often introduces an unnecessarily big search space and results in prolonged

running time. Reasonable alternatives are using either frequent subgraph patterns or discriminative subgraph patterns as candidates for transformation. GG-miner selects a set of most discriminative subgraph patterns as candidate GG-nodes for transformation, motivated by an empirical observation: If a pattern is frequent in the positive set, it is not necessarily a part of some larger discriminative subgraph patterns. But if a pattern is discriminative, it is much more likely to be a part of some larger and more discriminative patterns. Figure 6.2 and Figure 6.3 illustrate this observation made on a chemical graph database. Each data point in the two figures represents a 4-edge subgraph pattern  $p$ . In Figure 6.2, the x-coordinate of a pattern  $p$  is its positive frequency in the graph database. In Figure 6.3, the x-coordinate of a pattern  $p$  is its discrimination score in the database. In both figures, the y-coordinate of a pattern  $p$  is the average discrimination score of all the 8-edge patterns that contain  $p$ .



**Figure 6.2: Relationship between positive frequency and average discrimination score of super-patterns**



**Figure 6.3: Relationship between discrimination score and discrimination score of super-patterns**

It is clear that a high discrimination score is a much better indicator of being part of some larger discriminative patterns. Therefore, GG-miner selects discriminative subgraph patterns as candidate GG-nodes for graph transformation.

I further examined the correlation between the scores of small patterns and the scores of larger supergraph patterns as follows.

Given a pattern  $p$ , composed of  $n$  edges, consider a series of subgraphs  $p_1 \subset \dots \subset p_{n-1} \subset p$ , where  $p_i$  is a subgraph of pattern  $p$  with  $i$  edges for  $i \in [1, n-1]$ .  $P_p(p_i | p_{i-1})$  is defined as the conditional probability of a positive graph  $g$  containing  $p_i$  given that  $g$  contains  $p_{i-1}$ .

Similarly,  $P_n(p_i | p_{i-1})$  is defined for negative graphs. In addition,  $R(p_i, p_{i-1}) = \log \frac{P_p(p_i | p_{i-1})}{P_n(p_i | p_{i-1})}$

is defined to facilitate the analysis. The value of  $R(p_i, p_{i-1})$  indicates how important the extra edge in  $p_i$  is to the discrimination score of pattern  $p$ . The discrimination score of  $p$  can be expressed as:

$$score(p) = \log \frac{\prod_{i=1}^n P^+(p_i | p_{i-1})}{\prod_{i=1}^n P^-(p_i | p_{i-1})} = \sum_{i=1}^n \log R(p_i, p_{i-1})$$

The discrimination score of an  $m$ -edge subgraph  $p_m$  can be expressed as:

$$score(p_m) = \sum_{i=1}^m R(p_i, p_{i-1}) = score(p) - \sum_{i=m+1}^n R(p_i, p_{i-1})$$

The correlation between  $score(p_m)$  and  $score(p)$  depends on  $m$  and the deviation of  $R(p_i, p_{i-1})$ .

When the deviation of  $R(p_i, p_{i-1})$  is high, the value of some  $R(p_i, p_{i-1})$  can be large.

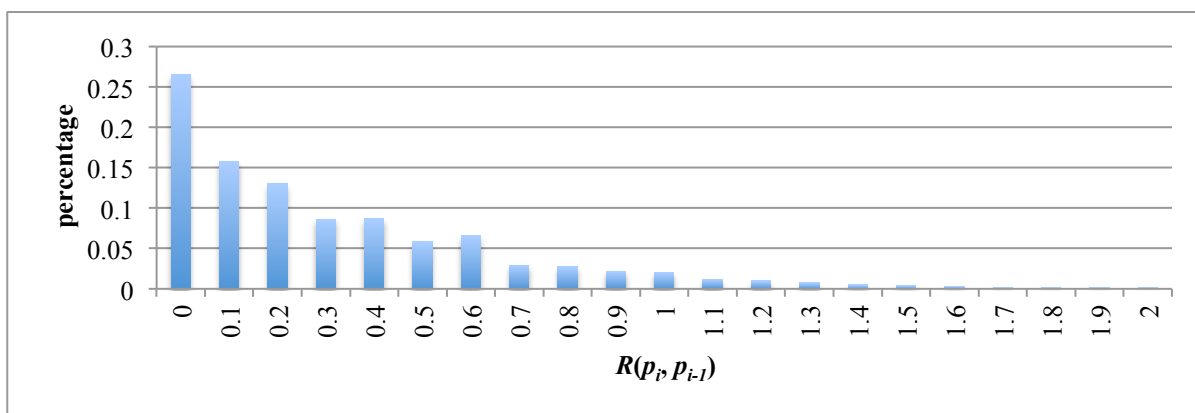
Thus,  $\sum_{i=m+1}^n R(p_i, p_{i-1})$ , which is the difference between  $score(p)$  and  $score(p_m)$ , can be very

large, especially when  $m$  is small. As a result, the correlation between  $score(p_m)$  and  $score(p)$  tends to be weak.

When the deviation of  $R(p_i, p_{i-1})$  is low, the value of each  $R(p_i, p_{i-1})$  is close to  $\frac{score(p)}{n}$ . When  $m$  is large,  $score(p_m)$  can be well approximated by  $score(p_m) \approx \frac{m}{n} score(p)$ .

As a result, the correlation between  $score(p_m)$  and  $score(p)$  tends to be strong.

Figure 6.4 shows the distribution of  $R(p_i, p_{i-1})$  for the chemical graphs used in Figure 6.2 and Figure 6.3. The average value of  $R(p_i, p_{i-1})$  is 0.372 and the standard deviation is 0.374. 64% of the  $R(p_i, p_{i-1})$  values are less than 0.4 and 88% are less than 0.7. Discrimination scores of discriminative subgraph patterns are typically greater than 2. Therefore, the deviation of  $R(p_i, p_{i-1})$  is low, which explains the strong correlation between the scores of small patterns and the scores of their larger supergraph patterns observed in Figure 6.2 and Figure 6.3. Similar observations are made in other chemical and protein datasets.



**Figure 6.4: Distribution of  $R(p_i, p_{i-1})$**

Low deviation in  $R(p_i, p_{i-1})$  is common in real datasets because higher deviation in  $R(p_i, p_{i-1})$  indicates less robustness in discriminative patterns. When the deviation is high, the discrimination power of a pattern mostly depends on a small number of edges because  $R(p_i, p_{i-1})$  indicates how important the extra edge in  $p_i$  is to the discrimination score of pattern  $p$ .

As a result, a mere edge mutation in a discriminative pattern can render the pattern indiscriminative. This is uncommon in practice.

#### 6.1.4 Phase 1: Discriminative Subgraph Mining and Graph Transformation

In phase 1, GG-miner searches for top- $k$  discriminative subgraphs for each positive graph and uses them to transform the original graphs to GG graphs.  $k$  is a user-specified parameter to adjust the trade-off between the efficiency of GG-miner and the discrimination power of the resulting patterns. A smaller  $k$  results in fewer candidates for transformation and thus smaller search space and higher efficiency, but it is more likely to miss highly discriminative patterns. A larger  $k$  results in larger search space, but it is more likely to find highly discriminative patterns.

Any discriminative subgraph mining algorithm can be adapted to mine the top- $k$  discriminative subgraphs for each positive graph. I used a variant of evolutionary computation of GAIA described as follows. Each positive graph  $g$  is associated with representative patterns that are the top- $k$  discriminative subgraphs found in this graph. The search begins with collecting all 1-edge subgraphs as candidate patterns and sorts them by their discrimination scores in decreasing order. For each candidate pattern, I enumerated all possible extensions to it and sort them by their discrimination scores. If the score of a pattern extension  $p'$  is not greater than any representative pattern at the time  $p'$  is generated, then  $p'$  is discarded; otherwise, the algorithm updates representative patterns and keeps  $p'$  for further extensions. The mining algorithm is described as follows.

**Algorithm:** *discriminative\_subgraph\_mining*( $G, k$ )  
 $G$ : positive graph set  $G_p$  and negative set  $G_n$   
 $k$ : the goal is to find the top- $k$  patterns for each positive graph  
 $top$ : maintains the top- $k$  patterns for each positive graph  
1.  $S = \{e \mid \exists g \in G_p, e \in E(g)\}$



2. **for each**  $e \in S$
3.      $subgraph\_extension(e)$

**Algorithm:**  $subgraph\_extension(p)$

$p$ : candidate subgraph to be extended

$G$ : positive graph set  $G_p$  and negative set  $G_n$

$k$ : the goal is to find the top- $k$  patterns for each positive graph

$top$ : maintains the top- $k$  patterns for each positive graph

1. **for each**  $g \in \{g \mid p \subseteq g, g \in G_p\}$
2.     **if**  $score(p) > score(top[g].worst)$
3.          $top[g].insert(p)$ ;
4. **for each**  $p' \in X = \{p' \mid p \subseteq p', |E(p')| = |E(p)| + 1\}$
5.     **if**  $score(p') > score(p)$
6.          $subgraph\_extension(p')$

During the mining process, GG-miner maintains the embeddings of all top- $k$  representative patterns. The embeddings of pattern  $p$  in graph  $g$  is denoted as  $EM(p, g)$ , which is a set of subgraphs in  $g$ . When the search terminates, GG-miner puts all these embeddings into one set  $P$  and transform each input graph  $g$  to  $GT(g, P)$ . The transformation algorithm is described below.

**Algorithm:**  $GG\_transform(g, P)$

$g$ : the graph to be transformed

$P$ : a set of subgraph patterns

$EM(p, g)$ : the embeddings of  $p$  in  $g$

$C$ : the set of GG-nodes of  $GT(g, P)$

$D$ : the set of GG-edges of  $GT(g, P)$

1.  $C \leftarrow \emptyset$ ;  $D \leftarrow \emptyset$ ;
2. **for each**  $p \in P$
3.     **for each**  $c \in EM(p, g)$
4.          $C = C \cup \{c\}$
5. **for each**  $a \in C$
6.     **for each**  $b \in C, a \neq b$
7.         **if**  $a.V \cap b.V \neq \emptyset$  **or**  $\exists (u \in a.V, v \in b.V): (u, v) \in g.E$
8.              $D = D \cup \{(a, b)\}$

### 6.1.5 Phase 2: Mining Transformed Graphs for GG-subgraph Patterns

In phase 2 GG-miner searches the GG graphs for discriminative GG-subgraph patterns. For each positive graph, GG-miner outputs the most discriminative GG-subgraph pattern found in the corresponding GG graph. When mining the GG graphs, GG-miner uses the same mining process as that used in mining original graphs in phase 1 with slight modification in initialization. The phase 2 mining process begins with all 1-GG-node patterns instead of 1-GG-edge patterns because, in GG graphs, a 1-GG-node pattern corresponds to patterns with multiple edges in the original graphs and thus can be highly discriminative.

The most discriminative GG-subgraph pattern in positive graph  $g$  found in phase 2 always have equal or better scores than the top- $k$  rigid subgraph patterns in  $g$  found in phase 1. This is because each rigid pattern found in phase 1 corresponds to a 1-GG-node GG-subgraph pattern in phase 2 and all 1-GG-node patterns are examined at the beginning of phase 2.

Experimental results show that phase 2 mining is much faster than phase 1 mining because (1) GG-nodes in the GG graphs are much more discriminative than nodes in the original graphs, which makes the pruning technique used in GG-miner much effective; (2) using discriminative subgraphs as GG-nodes in GG-graphs greatly reduces the number of edge extensions.

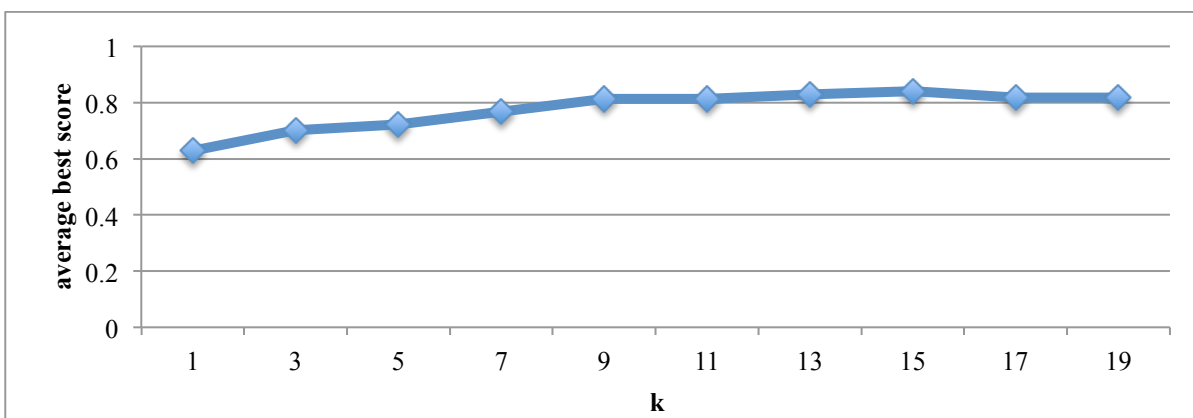
## 6.2 Experiments

I evaluated the performance of a discriminative substructure mining algorithm by its runtime efficiency and the best discrimination score of its resulting patterns. I first analyzed the effect of parameter  $k$  on the performance of GG-miner in 6.2.1. Then in 6.2.2, I compared GG-miner with COM, GAIA, LTS and LEAP in terms of pattern quality and runtime

efficiency. At last, in 6.2.3, I applied GG-miner to graph classification and compare its classification accuracy with COM, GAIA, LTS and graphSig.

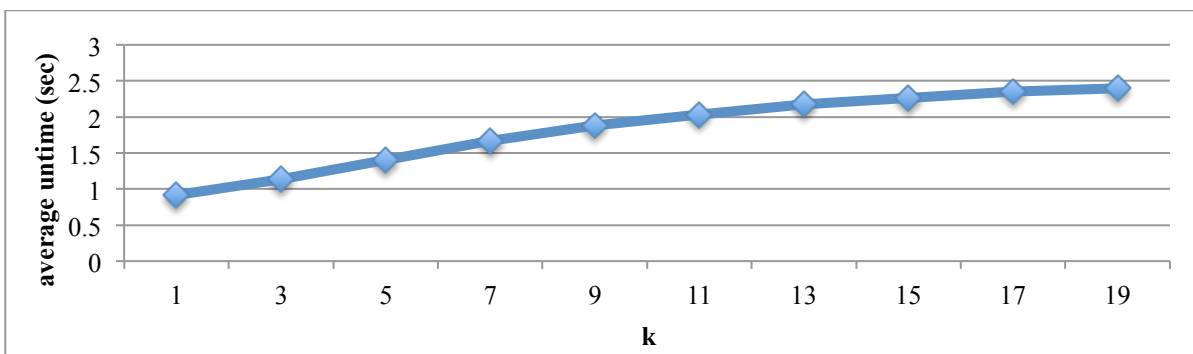
### 6.2.1 Performance Analysis

Figure 6.5 shows the effect of  $k$  on the average score of the best patterns found by GG-miner. In phase 1, GG-miner finds the top- $k$  discriminative subgraph patterns for each positive graph and uses their embeddings as candidate GG-nodes. When  $k$  increases, more structural information may be preserved in the resulting GG graphs. Consequently, GG-miner is able to find patterns with higher discrimination scores. However, the scores reach a saturation point after  $k$  exceeds 9.



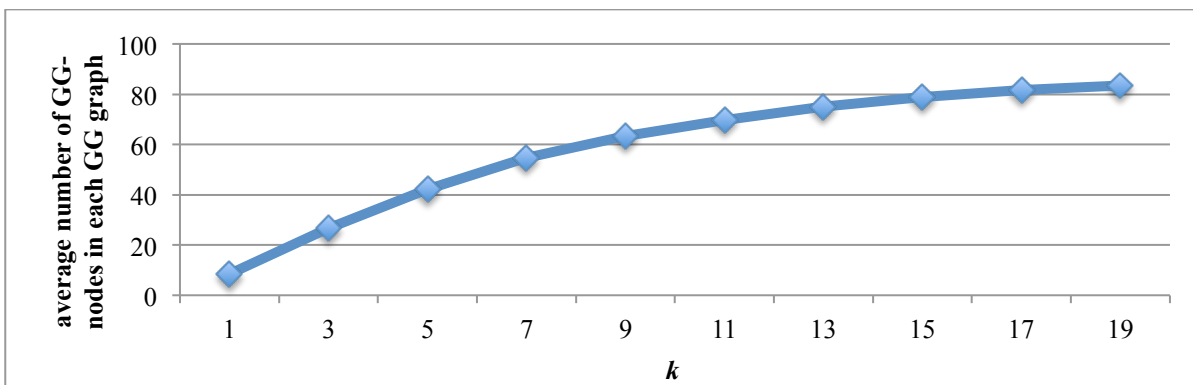
**Figure 6.5: Average best score vs.  $k$  (chemical datasets)**

Figure 6.6 shows that the runtime grows when  $k$  increases because there are more candidate GG-nodes to be enumerated in phase 1. In addition, the growth becomes slower as  $k$  increases. This is because it is easier to find patterns of lower discrimination scores.



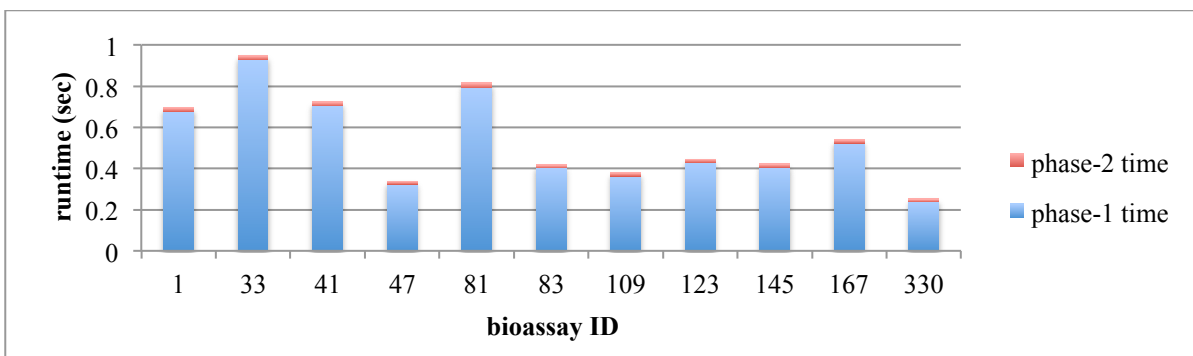
**Figure 6.6: Average runtime vs.  $k$  (chemical datasets)**

Figure 6.7 shows that the average number of GG-nodes in each GG graphs also increases as  $k$  increases. It can also be observed that the average number of GG-nodes will eventually reach a saturation point when  $k$  is large enough because the number of subgraphs in original graphs is finite. Additionally, the GG graphs become larger than the original graphs when  $k$  is greater than 7.



**Figure 6.7: Average number of GG-nodes in each GG graph vs.  $k$  (chemical datasets)**

Despite the increasing size of GG graphs, phase 1 remains more computationally demanding as suggested in Figure 6.8. On average, phase 2 consumes only 3.5% of the total runtime. This is because 1) GG-nodes in the GG graphs are much more discriminative than nodes in the original graphs and thus makes the pruning technique used in GG-miner much effective; 2) it requires fewer edge extension operations to generate substructure patterns in GG graphs than in original graphs.



**Figure 6.8: Runtime of phase-1 and phase-2 mining (chemical datasets)**

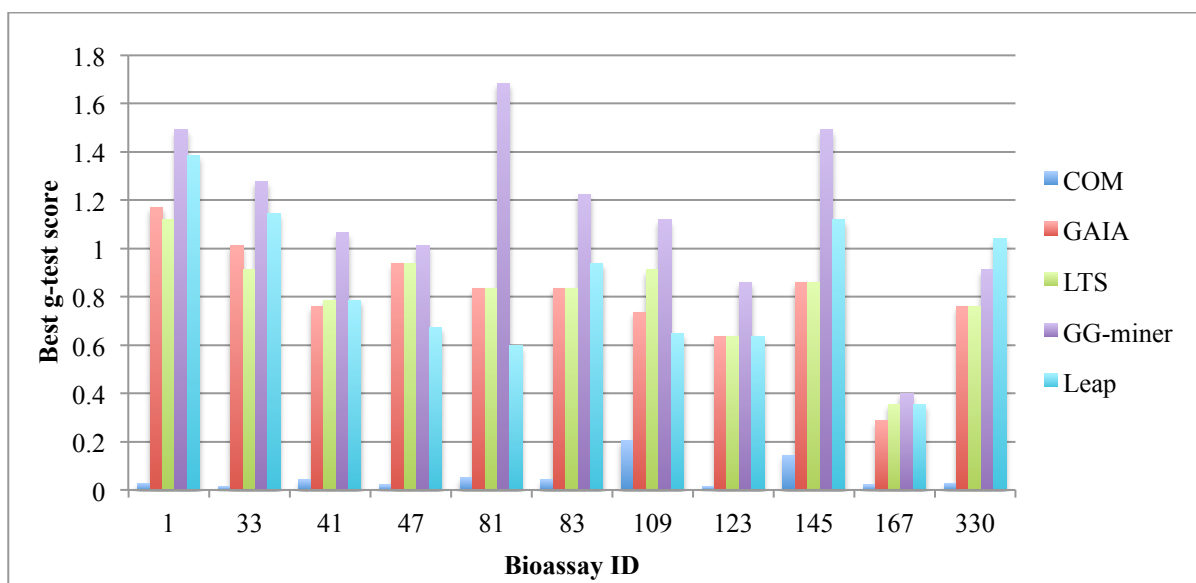
### 6.2.2 Pattern Quality and Runtime Efficiency Comparison

I compared the performance of GG-miner, COM, GAIA, LTS and LEAP in terms of runtime efficiency and the g-test scores of the best patterns found by the algorithms. LEAP uses leap length to balance between runtime efficiency and pattern quality. The larger the leap length is, the faster LEAP runs and the more likely it misses the optimal discriminative pattern. In the comparison for chemical datasets, the leap length was zero; in the comparison for protein datasets, the leap length was 0.05, which was the smallest LEAP could handle.

Figure 6.9 compares the g-test scores of the best patterns found by the algorithms in 11 chemical datasets. It shows that GG-miner finds better patterns than the other algorithms in 10 of 11 datasets. GG-miner has the highest average best score 1.14. LEAP is the second best algorithm in terms of discrimination power of resulting patterns and its average score is 0.847. After LEAP, LTS and GAIA have similar scores but the average score of LTS (0.813) is slightly higher than that of GAIA (0.803). COM has the worst average score 0.06.

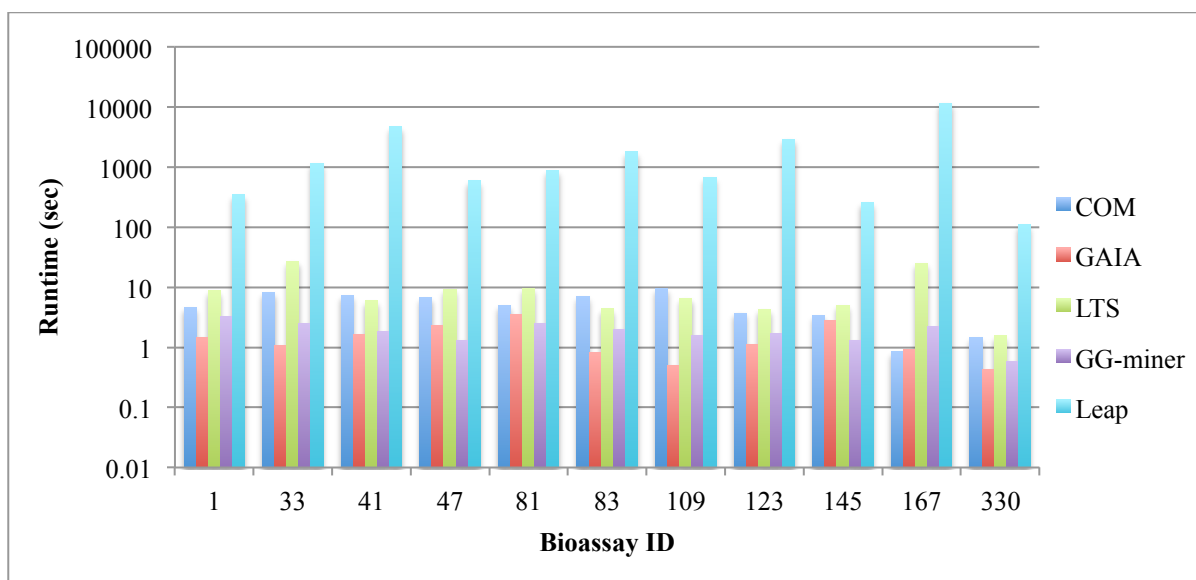
It demonstrates that it is worthy to allow flexibility in discriminative substructure pattern mining because it can lead to more discriminative patterns. Almost all of the best patterns found by GG-miner and LEAP have zero negative frequency. The difference in their scores results from their different positive frequencies. The best patterns found by GG-miner

have higher positive frequencies and thus better discrimination scores, thanks to the allowed flexibility.



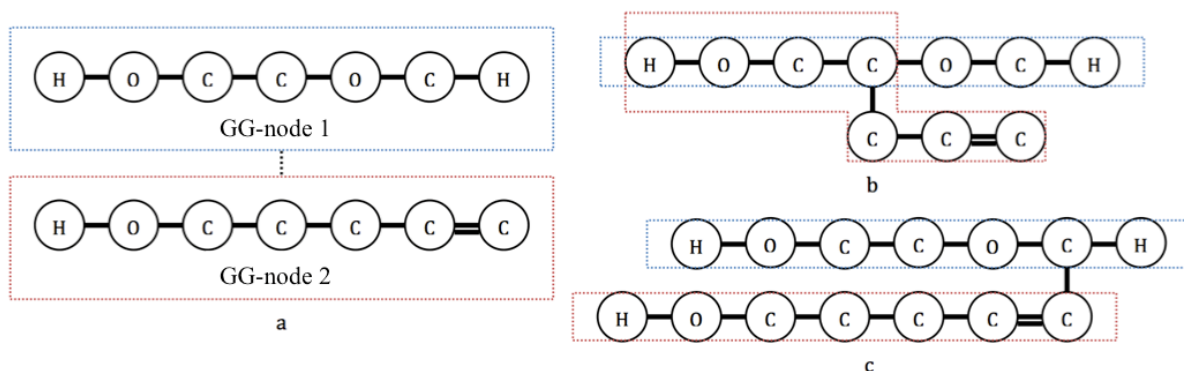
**Figure 6.9: Best g-test score comparison (chemical datasets)**

Figure 6.10 shows that LEAP is the slowest algorithm in all chemical datasets. and it takes 2265 seconds to process one dataset. On average, GAIA is the fastest and its average runtime is 1.5 seconds. GG-miner is the second fastest and its average runtime is 1.88 seconds. It is slightly slower than GAIA, but its pattern quality is significantly higher than GAIA, as can be seen in Figure 6.9. After GG-miner, the average runtimes of COM and LTS are 5.25 seconds and 9.72 seconds, respectively.



**Figure 6.10: Runtime comparison (chemical datasets)**

Figure 6.11 shows the most discriminative GG-subgraph pattern found by GG-miner in bioassay 1 to illustrate the flexibility in patterns found by GG-miner. It has two GG-nodes and one GG-edge. This pattern is found in 15.5% of the positive graphs and none of the negative graphs. Figure 6.11 (b) (found in 12% of the positive graphs) and (c) (found in 3.5% of the positive graphs) are various appearances of the same pattern in Figure 6.11 (a) in the positive set. Both appearances include the same two GG-nodes with different connections. In Figure 6.11 (b), the two GG-nodes overlap; in Figure 6.11 (c), the two GG-nodes are connected directly through a single bond between two carbons. They are considered as variants of the same pattern because GG-miner allows structural flexibility in how GG-nodes are connected in GG graphs.



**Figure 6.11: A pattern (figure a) in the GG graphs and its various appearances (figures b and c) in the original graphs**

Figure 6.12 compares the g-test scores of the best patterns found by COM, GAIA, LTS, GG-miner and LEAP in the protein datasets. The score of the best pattern found by GG-miner is always greater than or equal to the score of the best pattern found by other algorithms except for SCOP family 46463. The average best score found by GG-miner is 8.47. The second best algorithm for the protein datasets is LTS and its average score is 7.45. The average scores of COM, GAIA and LEAP are 5.15, 7.06 and 5.55, respectively.

Among the five algorithms, GAIA is the fastest, averaging 3.45 seconds. LTS is slightly slower and its average runtime is 3.72 seconds. The average runtimes of COM, GG-miner and LEAP are 5.7, 10.19 and 420.98 seconds, respectively.



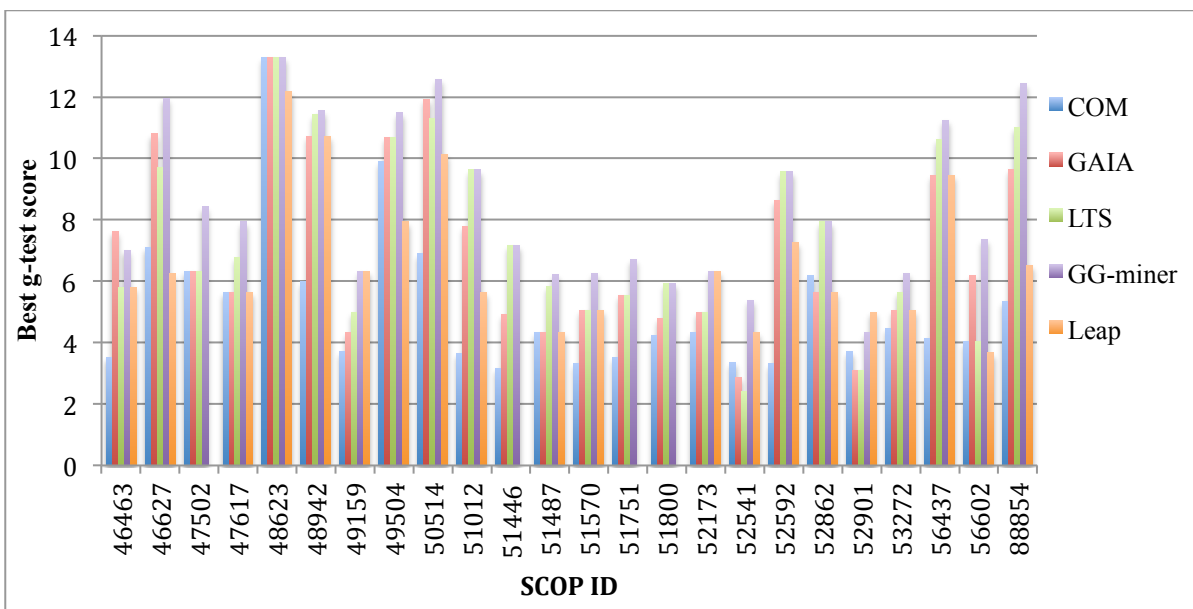


Figure 6.12: Best g-test score comparison (protein datasets)

### 6.2.3 Application in Graph Classification

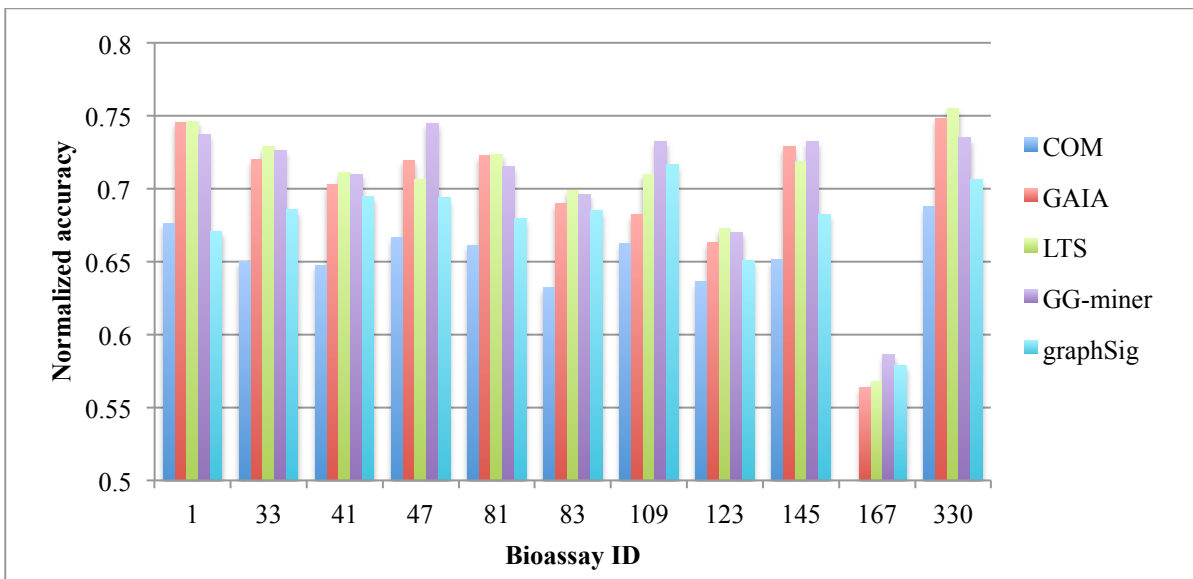
I applied GG-miner to graph classification in the SCOP protein datasets and PubChem chemical compound datasets and compared its classification accuracy and runtime efficiency with COM, GAIA, LTS and graphSig. All the graph classification experiments are performed with 5-fold cross-validation. The parameter settings for COM, GAIA and graphSig are summarized in Table 6.1.

Table 6.1: Parameter settings of GAIA and graphSig

	Proteins	Chemicals
GAIA	list size = 100 iteration # = 10 CPU # = 32	list size = 10 iteration # = 4 CPU # = 1
graphSig	N/A	maxPvalue=0.1 minFreq=0.1%
COM	$t_p=30\%$ , $t_n=0\%$	$t_p=1\%$ , $t_n=0.4\%$

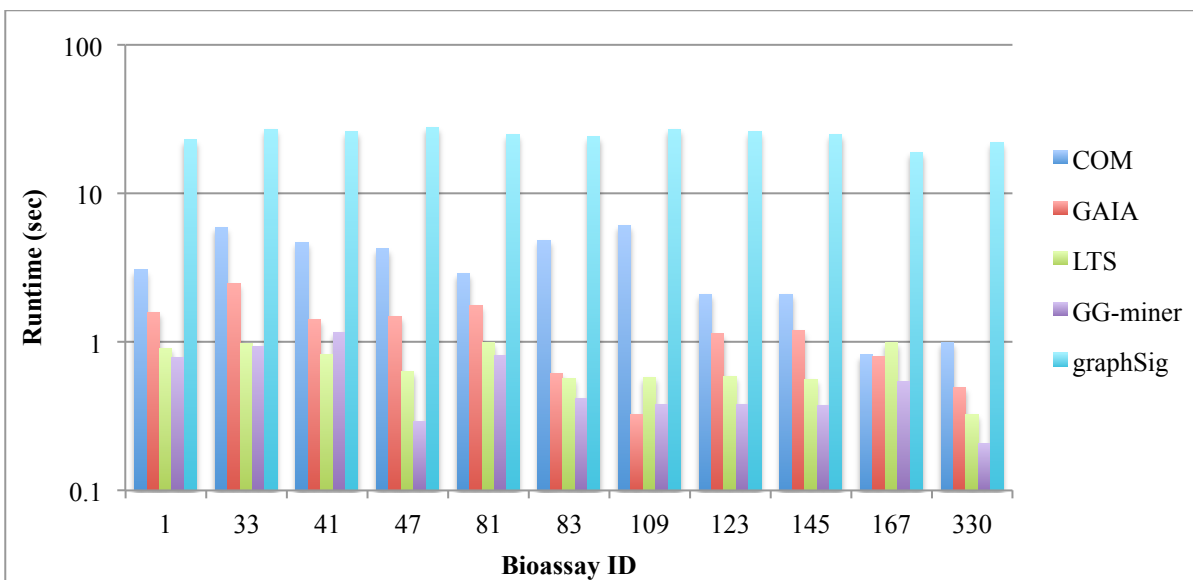
Figure 6.13 illustrates the normalized accuracy achieved by the five algorithms for chemical datasets. GAIA, LTS and GG-miner have very similar performance in normalized

accuracy. The average accuracies of GAIA, LTS and GG-miner are 70.3%, 70.4% and 70.8%. LTS has the highest accuracy in 7 out of 11 datasets and GG-miner has the highest accuracy in 4 out of 11 datasets. COM and graphSig typically have considerably lower normalized accuracy than the other three. The average accuracy of COM is 64.3% and the average accuracy of graphSig is 67.7%.



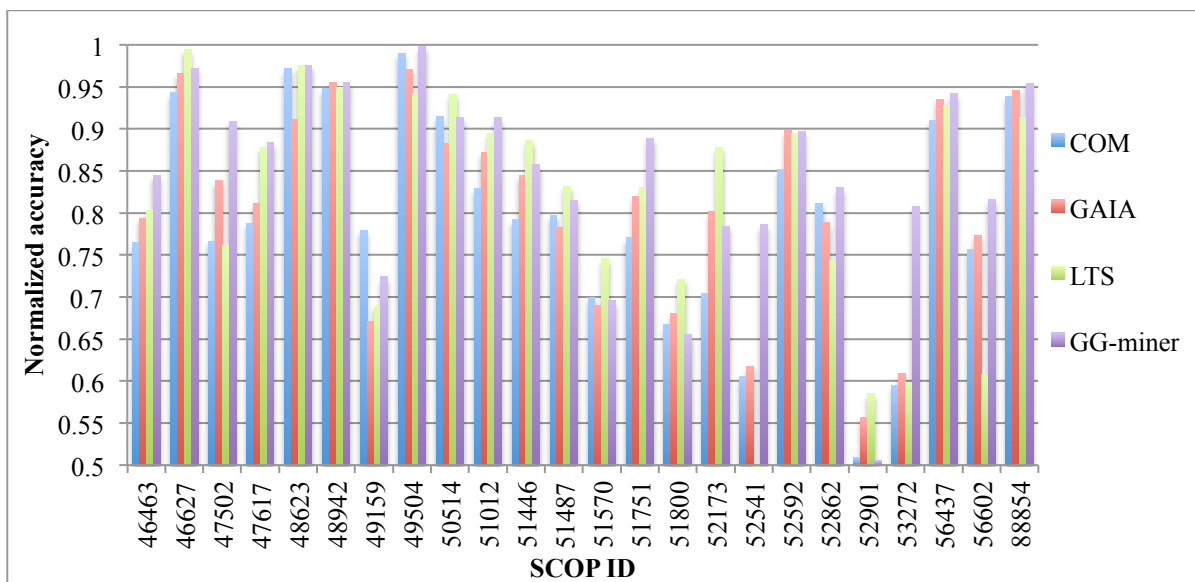
**Figure 6.13: Normalized accuracy comparison (chemical datasets)**

Figure 6.14 compares the runtime of the five algorithms. It shows that graphSig is the slowest algorithm among the five, averaging 24.7 seconds. GG-miner is the fastest and its average runtime is 0.57 seconds. The average runtimes of COM, GAIA and LTS are 3.43, 1.30 and 0.72 seconds, respectively.



**Figure 6.14: Runtime comparison (chemical datasets)**

I also ran the algorithms on the protein datasets as a supplementary comparison and reported the accuracy result in Figure 6.15. The average accuracy of GG-miner (84.7%) is 5.1% higher than that of COM (79.6%), 3.8% higher than that of GAIA (80.9%) and 3.5% higher than that of LTS. GG-miner achieves the highest classification accuracy, but its average runtime is longer than COM, GAIA and LTS. The average runtime of GG-miner is 7.46 seconds. The average runtimes of COM, GAIA and LTS are 2.79, 2.63 and 3.27, respectively.



**Figure 6.15: Normalized accuracy comparison (protein datasets)**

## **Chapter 7**

### **Mining Discriminative Subgraph Patterns in Graphs with Continuous Label Values**

A common problem faced by previous discriminative subgraph pattern mining algorithms is that they are designed for graphs with discrete label values and are thus unable to process graphs with continuous label values directly. For example, in protein graphs, each node represents an amino acid and each edge can be labeled with distances between corresponding amino acids, which are continuous values.

In previous studies [Bandyopadhyay2006, Fei2010, Huan2003, Huan2004, Huan2009, Jin2009, Jin2010, Thoma2009], this problem of being unable to process continuous label values was addressed by arbitrarily discretizing continuous values into bins. The discretization scheme can affect the effectiveness of the discriminative subgraph mining. A fine discretization with more bins preserves more information from continuous values in the original graphs and enables subgraph patterns to be more informative. However it is more susceptible to noise and error in the original graphs and disallows flexible subgraph patterns due to the rigid label matching requirement in subgraph isomorphism. On the contrary, a coarse discretization with fewer bins loses some accuracy of the original graphs and often results in subgraph patterns that are less expressive, but it is less vulnerable to noise and error and also allows flexibility in subgraph patterns. Besides, any discretization scheme is

susceptible to the artifact of the boundary effects of bins. These factors impair the quality of the subgraph patterns found by the current discriminative subgraph mining algorithms.

In this chapter, I proposed a solution to overcome the discretization problem by performing edge relaxation on discriminative subgraph patterns with discretized edge labels and I demonstrated that the proposed solution produces subgraph patterns with higher discrimination power. In this solution, I adapted GAIA [Jin2010] to first generate candidate discriminative subgraphs and then perform edge relaxation on the candidate patterns. However, the edge relaxation operation can be performed on discriminative subgraphs found by any algorithm.

The solution concentrates on graphs whose edges have continuous labels. However, it is straightforward to extend the idea to process graphs whose nodes also have continuous labels.

## 7.1 Edge Relaxation

I defined a relaxed subgraph pattern generated by edge relaxation as  $(V', E', \Delta)$ , where  $V'$  is a set of nodes,  $E'$  is a set of edges describing the expected distances between the residues in  $V'$  and  $\Delta$  is an RMSD (Root Mean Squared Deviation) threshold. This model can be considered as a graph (with a node set  $V'$  and an edge set  $E'$ ) with a parameter  $\Delta$  specifying the allowed amount of relaxation.

Given a subgraph pattern  $p$  with discrete edge labels, a positive set  $G_p$  and a negative set  $G_n$ , to generate the corresponding relaxed subgraph pattern  $p'$  with continuous edge labels, the edge relaxation algorithm first finds all the occurrences of  $p$  in  $G_p$  and for each edge  $(u, v)$  in  $p$  the algorithm calculates the average edge label value between nodes  $u$  and  $v$

in the occurrences. This average edge label value is used to label the corresponding edge  $(u, v)$  in the relaxed subgraph pattern  $p'$ . The node set of  $p'$  is the same as that of  $p$ .

The second step of edge relaxation is to compute the RMSD threshold  $\Delta$ . Given a subgraph pattern  $p'$  with continuous edge labels and a graph  $g$  with continuous edge labels, I defined the RMSD between  $p'$  and  $g$  as follows:

$$RMSD(p', g) = RMSD(p', g, \underset{f: V(p') \rightarrow V(g), \forall u \in V(p'), f(u) = l(f(u))}{\operatorname{argmin}} (RMSD(p', g, f)))$$

or  $+\infty$  when  $f$  does not exist, where

$$RMSD(p', g, f) = \sqrt{\frac{1}{|E(p')|} \sum_{(u,v) \in E(p')} (1 - \frac{l((f(u), f(v)))}{l((u, v)))^2}$$

The intuition is to find an approximate embedding of subgraph pattern  $p'$  with continuous edge labels in graph  $g$  (ignoring edge labels) and then calculate the RMSD between  $p'$  and the embedding. When the edge relaxation algorithm calculates the RMSD, it normalizes the continuous edge labels in  $g$  by the corresponding continuous edge labels in  $p'$ . There can be more than one approximate embedding of  $p'$  in  $g$  and the one with the minimum RMSD is selected and the corresponding RMSD is used as the RMSD between  $p'$  and  $g$ .

The RMSD threshold  $\Delta$  is used to specify how much relaxation is allowed for a relaxed subgraph pattern  $p'$ . I defined that: if  $RMSD(p', g)$  is less than  $\Delta$ , then the graph  $g$  contains  $p'$ ; otherwise,  $g$  does not contain  $p'$ . Given the RMSD threshold  $\Delta$  for a relaxed subgraph pattern  $p'$ , the frequencies and discrimination score of  $p'$  in the positive set and negative set are defined as follows:

$$pfreq(p', \Delta) = \frac{|\{g \mid g \in G_p, RMSD(p', g) < \Delta\}|}{|G_p|}$$

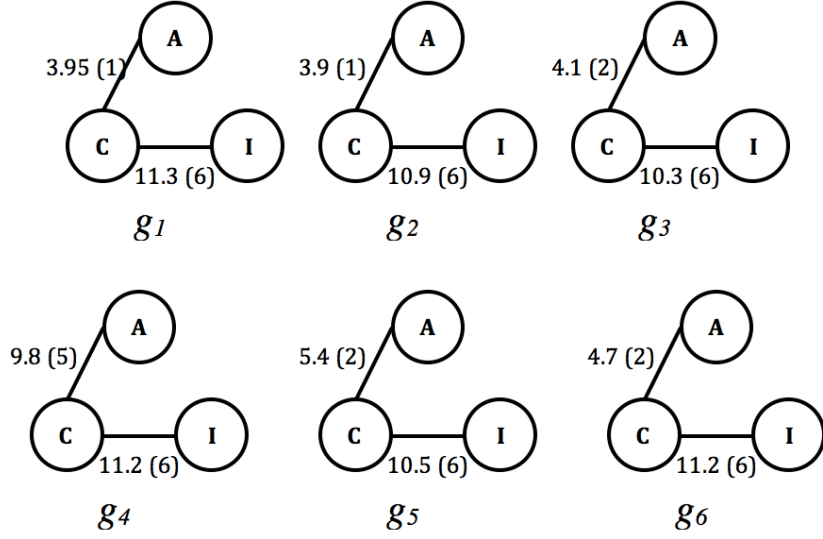
$$nfreq(p', \Delta) = \frac{|\{g \mid g \in G_n, RMSD(p', g) < \Delta\}|}{|G_n|}$$

The edge relaxation algorithm computes the value of  $\Delta$  for each relaxed discriminative subgraph pattern  $p'$  to optimize its discrimination score in the input graph dataset and the margins between  $\Delta$  and the nearest RMSD values:

$$\Delta = \arg \max_{\Delta' \in \{d \mid d = (RMSD(p', g_1) + RMSD(p', g_2))/2, g_1 \in G_p, g_2 \in G_n\}} (score(p', \Delta'))$$

I illustrated the edge relaxation process with an example in Figure 7.1. The positive set is composed of graphs  $g_1$ ,  $g_2$  and  $g_3$ ; the negative set consists of graphs  $g_4$ ,  $g_5$  and  $g_6$ . Each edge in a graph is labeled with both the continuous label value and the discretized label (in parentheses). Let the subgraph pattern  $p$  found by the discriminative mining algorithm be A-1-C-6-I, which occurs only in  $g_1$  and  $g_2$ . The goal is to generate a relaxed discriminative subgraph pattern  $p'$  based on  $p$  using edge relaxation. The first step is to compute the average label values for edges in this subgraph and the result is A-3.925-C-11.1-I (only its two occurrences are considered in this step), which is the graph description of the new relaxed subgraph pattern  $p'$ . Then the edge relaxation algorithm computes the RMSD between A-3.925-C-11.1-I and each graph. The results are: 0.008, 0.008, 0.07, 1.06, 0.27 and 0.14 (for  $g_1$ ,  $g_2$ ,  $g_3$ ,  $g_4$ ,  $g_5$ ,  $g_6$ ). The last step is to compute  $\Delta$  for the new pattern  $p'$ . The value of  $\Delta$  is set to maximize the score of  $p'$  and the margins between  $\Delta$  and the nearest RMSD values. In the example, any value of  $\Delta$  in the range (0.07, 0.14] can optimize the score of  $p'$ . In order to maximize the margins between  $\Delta$  and the nearest RMSD values, the edge relaxation algorithm sets  $\Delta$  as the medium value 0.105 of this range. Thus, the relaxed discriminative subgraph pattern is A-3.925-C-11.1-I with  $\Delta = 0.105$ .





**Figure 7.1: An example to illustrate edge relaxation**

## 7.2 Experiments

I designed and implemented an algorithm named MSG to mine relaxed discriminative subgraph patterns. It first discretizes continuous edge labels. For protein graphs, I used the following discretization scheme: 0-4Å, 4-5.5Å, 5.5-7Å, 7-8.5Å, 8.5-10Å and 10-11.5Å. Then the algorithm MSG invokes single-GAIA to search for rigid discriminative subgraph patterns as candidates for edge relaxation. In the end, MSG performs edge relaxation on the candidate patterns to improve their discrimination power and selects the best relaxed patterns to output as results.

I used all SCOP [Murzin1995] families with at least 20 proteins to evaluate the performance of the proposed method MSG.

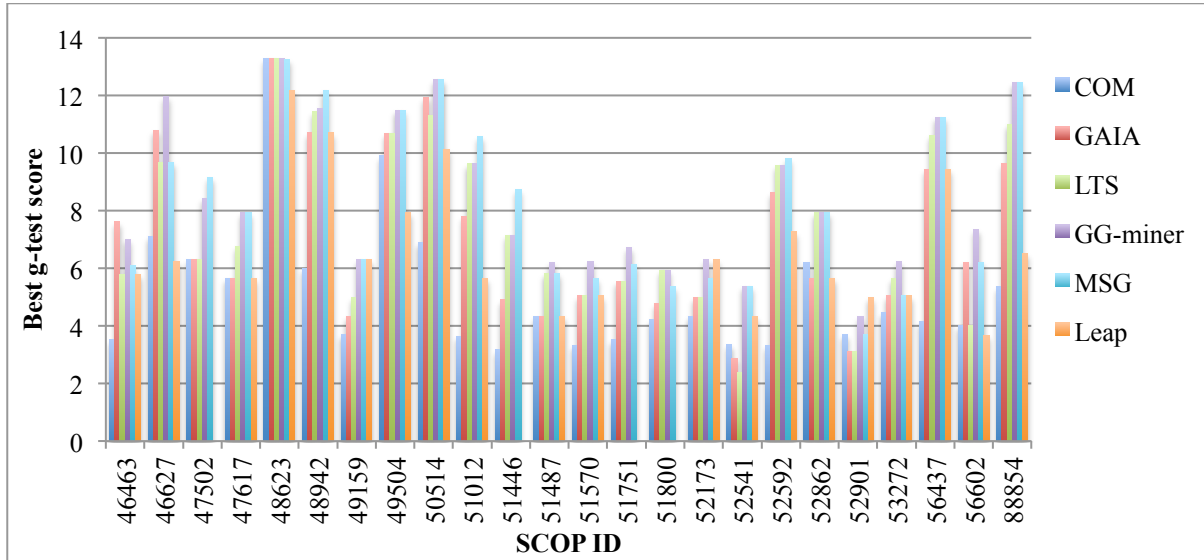
### 7.2.1 Quality of Relaxed Subgraph Patterns

I studied the quality of relaxed subgraph patterns found by MSG through comparison with other methods. I ran MSG (single GAIA with edge relaxation), COM ( $t_p=30\%$ ,  $t_n=0\%$ ), LTS, LEAP (leap length = 0.05) and GAIA-32 (parallel-GAIA with 32 CPUs) respectively

to find the most discriminative patterns and reported the runtime and the discrimination scores (the higher the better) of the optimal pattern found by each method.

The best g-test scores found by the algorithms in each family are illustrated in Figure 7.2. In 23 out of 24 SCOP families, MSG finds subgraph patterns with better discrimination scores than COM, GAIA, LTS and LEAP. In 14 out of 24 SCOP families, MSG finds subgraph patterns with better discrimination scores than GG-miner. However, GG-miner has the highest average score (8.47), which is slightly higher than the average score of MSG (8.27). The average scores of COM, GAIA, LTS and LEAP are 5.15, 7.06, 7.45 and 5.55, respectively.

This comparison demonstrates the advantage of MSG over rigid subgraph mining algorithms, such as COM, GAIA, LTS and LEAP, in terms of the quality of subgraph patterns. Although MSG does not outperform GG-miner because both allow flexibility in patterns, the two algorithms can complement each other as they concentrate on different types of flexibility.



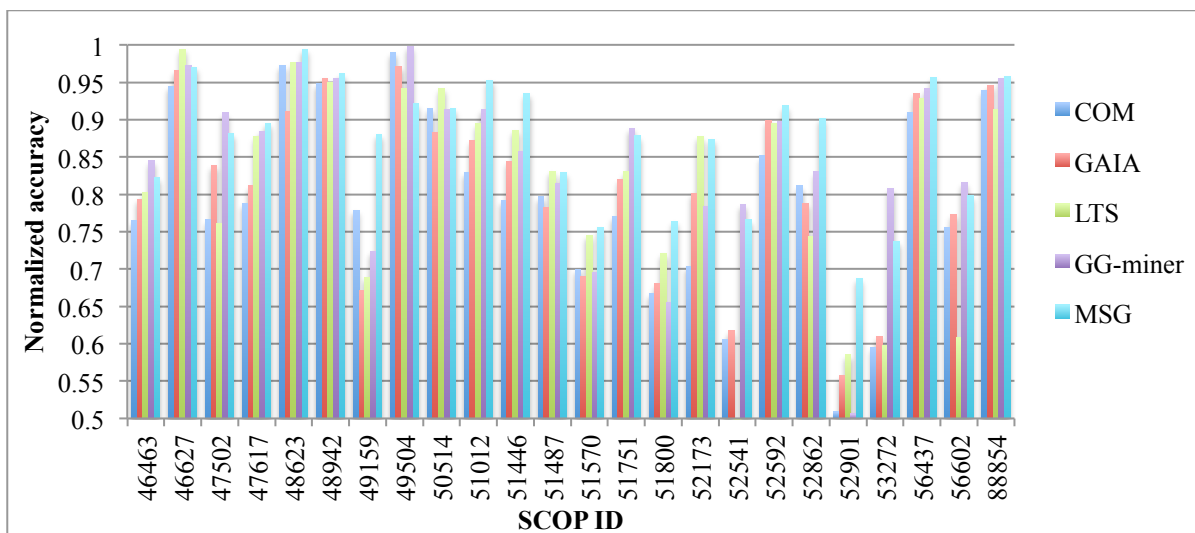
**Figure 7.2: Best g-test score comparison**

As to runtime performance, MSG is the second slowest, averaging 23.4 seconds. On average, GAIA is the fastest (3.45 seconds) and LEAP is the slowest (420.98 seconds). The average runtimes of COM, LTS and GG-miner are 5.7, 3.72 and 10.19 seconds, respectively.

### 7.2.2 Classification Accuracy

I performed binary protein classification using the subgraph patterns found by COM, GAIA, LTS, GG-miner and MSG to illustrate what benefit the higher quality of relaxed patterns found by MSG can provide.

The normalized accuracy of each method for each family is illustrated in Figure 7.3. MSG achieves higher accuracy than the other four proposed algorithms in 14 out of 24 SCOP families. On average, the normalized accuracy of MSG (87.3%) is 7.7% higher than the accuracy of COM (79.6%), 6.4% higher than the accuracy of GAIA (80.9%), 6.1% higher than the accuracy of LTS (81.2%) and 2.6% higher than the accuracy of GG-miner (84.7%). It demonstrates that the relaxed discriminative subgraph patterns found by MSG are more effective in protein classification than those found by the other proposed algorithms.

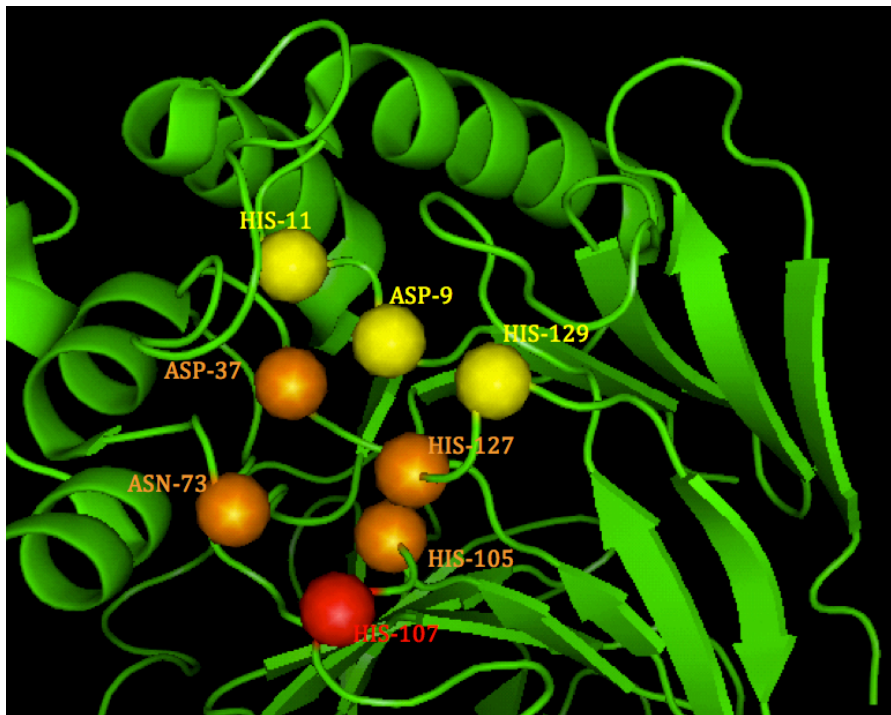


**Figure 7.3: Normalized accuracy comparison**

### 7.2.3 Case Study 1: Active Site Identification

The structure of protein YfcE (PDB: 1su1A) was determined and deposited in Protein Data Bank (PDB) as a structural genomic target of unknown function (until January 62009). However, there is experimental evidence showing that this protein has metallo-dependent phosphatase (Metallophos) activity [Miller2007]. From the Pfam [Finn2010] Metallophos family, I selected 9 proteins as the positive training set (1s95A, 1g5bA, 1s70A, 1kbpA, 1ii7A, 1au1A, 1xzwA, 1uteA and 1qhwa) such that their pairwise sequence identities are less than 90%, their sequence identities to 1su1A are all less than 20% and their DALI Z-scores [Holm2010] to 1su1A are all less than 14. This Pfam family corresponds to the SCOP superfamily 56300 and thus it is more diverse than a SCOP family. I randomly selected 1000 proteins outside the Pfam family as the negative training set. Holm et al. [Holm2008] suggested that a strong match for function annotation should have sequence identity above 20% or a Z-score above  $n/10-4$  where  $n$  is the number of residues (the Z-score cutoff for 1su1A is 14.4 as it has 184 residues) in the query structures, therefore the training set I used is insufficient to infer function annotation for 1su1A using either sequence alignment or

global structural alignment. However, the proposed method MSG can find highly discriminative subgraph patterns from the training set. All of these patterns are present in 1su1A, which leads to a reliable function annotation. In addition, more than half of the residues in the active site of 1su1A are covered by the best pattern found by MSG.



**Figure 7.4: Positions of the active site and the best pattern found by MSG in 1su1A.**

The top 5 patterns found by MSG are listed in Table 7.1. The top 3 patterns found by MSG occur in all 9 proteins in the family but are absent in every background proteins. The 4<sup>th</sup> and 5<sup>th</sup> patterns occur in 8 family proteins and are absent from the background proteins. All five patterns are present in 1su1A, so with high confidence it can be inferred that 1su1A has the same function as the family proteins. One of the top patterns includes 4 residues of the active site (composed of 7 residues: ASP-9, HIS-11, ASP-37, ASN-73, HIS-105, HIS-127 and HIS-129) of 1su1A (illustrated in Figure 7.4) and the other two top patterns include

3 residues of the active site respectively. The 4<sup>th</sup> and 5<sup>th</sup> pattern each covers 2 residues of the active sites respectively. The top 5 patterns together cover 6 out of 7 residues of the active site. Each of the top 3 patterns covers 5 residues. The 4<sup>th</sup> pattern covers 4 residues and the 5<sup>th</sup> pattern covers 5 residues. It can be seen that the relaxed discriminative subgraph patterns found by MSG can not only predict the function of 1su1A but indicate the position of the active site as well.

**Table 7.1: Top-5 relaxed discriminative subgraph patterns found by MSG**

Pattern ID	positive support	negative support	The occurrence with the minimal RMSD in 1su1A (residues involved in the active site of 1su1A are in bold and italic fonts)
1	9	0	<i>ASP-37, ASN-73, HIS-105, HIS-127</i> , HIS-107
2	9	0	<i>ASN-73, HIS-105, HIS-127</i> , HIS-107, LEU-130
3	9	0	<i>ASN-73, HIS-105, HIS-127</i> , HIS-107, GLY-126
4	8	0	<i>HIS-11, ASP-37</i> , GLY-12, ASN-40
5	8	0	<i>ASP-9, HIS-11</i> , GLY-12, SER-13, ASN-40

#### 7.2.4 Case Study 2: Function Inference

There are evidences showing that protein 1m65A belongs to the superfamily of metallo-dependent hydrolases, but discriminative subgraph patterns found by rigid subgraph mining with rigid edge labels in the superfamily are absent in 1m65A because of the high flexibility of the underlying active sites [Bandyopadhyay2009]. I used MSG to search for relaxed discriminative subgraph patterns in the same superfamily (SCOP ID = 51556, 23 members, excluding 1m65A) with a negative set composed of all other proteins in SCOP with less than 90% pair-wise sequence identity. Then I located the top-10 patterns in 1m65A. Before edge relaxation, only one of the top-10 patterns found by MSG are present in 1m65A (the pattern is found in 34.8% of the positive set and 0% of the negative set); after edge relaxation, three of the top-10 patterns are present in 1m65A (the patterns are found in

56.5%, 39% and 39% of the superfamily respectively and none of them is found in the negative set). Not only does edge relaxation leads to more discriminative subgraph patterns present in 1m65A but it leads to higher confidence of the function inference as well because of the higher frequency in the superfamily.

## Chapter 8

### Conclusions and Future Directions

#### 8.1 Conclusions

Discriminative subgraph patterns can be used to identify feature substructures and perform structure classification. Many research studies have been devoted to developing efficient and effective discriminative subgraph mining algorithms. Higher efficiency allows users to process larger graph datasets and higher effectiveness enables users to achieve better results in applications including protein classification, protein active site identification and chemical compound activity prediction.

Various techniques to improve efficiency and effectiveness are proposed and evaluated in this dissertation. Experimental results show that the proposed algorithms outperform other algorithms in terms of both efficiency and effectiveness. Table 8.1 summarizes the performance of proposed algorithms and major competitor algorithms. Bold font indicates best performance in the row.



**Table 8.1: Performance summary**

		GAIA	LTS	GG-miner	MSG	LEAP	graphSig
protein datasets	runtime (sec)	<b>2.63</b>	3.27	7.46	15.4	421	N/A
	accuracy	80.9%	81.2%	84.7%	<b>87.3%</b>	80.2%	N/A
	best score	7.06	7.45	<b>8.47</b>	8.27	5.55	N/A
chemical datasets	runtime (sec)	1.21	0.720	<b>0.572</b>	N/A	62.9	24.7
	accuracy	69.9%	70.4%	<b>70.8%</b>	N/A	67%	67.7%
	best score	0.803	0.813	<b>1.14</b>	N/A	0.847	N/A

Below I review some plausible future directions for discriminative subgraph mining.

## 8.2 Future Directions

### Learning from Search History with Multi-task Learning

The upper-bound estimation in LTS is susceptible to outliers because the sample size is small. In addition, the sample search space may not be representative enough due to its small size. One possible future direction is to investigate the feasibility of using multi-task learning to overcome these problems. Multi-task learning mines multiple related target datasets at a time. The target datasets are assumed to share similar search spaces and thus algorithms are able to merge the small sample search spaces from each target dataset into a large sample space. When the sample size is large, algorithms can also assign a confidence value to each upper-bound estimation to deal with outliers.

### Handling Nodes with Multiple Labels

In existing discriminative subgraph mining algorithms, each graph node is assumed to have only one label to describe the attribute of the node. However, it may not be the case in

some applications. For example, in social network graphs, each person corresponds to one graph node but each person can have multiple attributes. Although new labels can be created to represent all possible combinations of attribute values, this solution is infeasible when the number of attributes and possible attribute values is large. In addition, when each person has many attributes, it is unlikely that two persons share exactly the same values for all attributes. As a result, existing subgraph mining algorithms will fail to find any meaningful pattern. Therefore, it is necessary to develop novel algorithms that are capable of handling nodes with multiple labels. The new algorithms should be able to select node labels in order to identify discriminative subgraph patterns.

### **Improving Pattern Robustness in Noisy Data**

The input of discriminative subgraph pattern mining contains two sets of graphs and the algorithm assumes the input graphs are correctly classified into the two sets. However, the assumption does not hold when there is error or ambiguity in the classification of the input graphs. Besides, in my study I discover that discriminative subgraph mining algorithms are susceptible to overfitting and they can find discriminative subgraph patterns in randomly classified graphs. As a result, discriminative subgraph patterns will overfit input data with false positives/negatives and fail to capture the true underlying feature substructures. Therefore, there is a need for novel robust discriminative subgraph pattern mining algorithms that can tolerate a reasonable amount of misclassification in input.

## References

- [Bandyopadhyay2006] D. Bandyopadhyay, J. Huan, J. Liu, J. Prins, J. Snoeyink, W. Wang, and A. Tropsha. Structure-based function inference using protein family-specific fingerprints, *Protein Science*, vol. 15, pp. 1537-1543, 2006.
- [Bandyopadhyay2009a] Bandyopadhyay D, Huan J, Prins J, Snoeyink J, Wang W, Tropsha A. Identification of family-specific residue packing motifs and their use for structure-based protein function prediction: I. Method development. *J Comput Aided Mol Des* 2009.
- [Bandyopadhyay2009b] Bandyopadhyay D, Huan J, Prins J, Snoeyink J, Wang W, Tropsha A. Identification of family-specific residue packing motifs and their use for structure-based protein function prediction: II. Case studies and applications. *J Comput Aided Mol Des* 2009.
- [Chen2006] Chen, B., Fofanov, V., Bryant, D., Dodson, B., Kristensen, D., Lisewski, A., Kimmel, M., et al. (2006). Geometric sieving: Automated distributed optimization of 3D motifs for protein function prediction. In *RECOMB* (pp. 500–515).
- [Chen2008] Wen-Yen Chen, Dong Zhang, Edward Chang. Combinational Collaborative Filtering for Personalized Community Recommendation. *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2008, pp. 115-123.
- [Fei2008] Fei H and Huan J. Structure Feature Selection For Graph Classification. *ACM 17th International Conference of Knowledge Management 2008 (CIKM08)*. Napa Valley, California 2008.
- [Fei2010] Fei H and Huan J, Boosting with Structure Information in the Functional Space: an Application to Graph Classification, *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2010.
- [Finn2010] R.D. Finn, J. Mistry, J. Tate, P. Coghill, A. Heger, J.E. Pollington, O.L. Gavin, P. Gunasekaran, G. Ceric, K. Forslund, L. Holm, E.L. Sonnhammer, S.R. Eddy, A. Bateman. The Pfam protein families database. *Nucleic Acids Research* (2010) Database Issue 38:D211-222.
- [Fröhlich2005] Fröhlich H, Wegner J.K., Sieker F, Zell A. Optimal Assignment Kernels for Attributed Molecular Graphs, in *Proceedings of the 22nd International Conference on Machine Learning (ICML)*, pp. 225-232, 2005.
- [Helma2004] C. Helma, T. Cramer, S. Kramer, and L.D. Raedt. Data mining and machine learning techniques for the identification of mutagenicity inducing substructures and structure activity relationships of noncongeneric compounds. *J. Chem. Inf. Comput. Sci.*, 44:1402-1411, 2004.
- [Holm2008] L. Holm, S. Kaariainen, P. Rosenstrom and A. Schenkel, Searching protein structure databases with DaliLite v.3. *Bioinformatics* 24, 2780-2781 (2008).

- [Holm2010] Holm L, Rosenström P (2010) Dali server: conservation mapping in 3D. *Nucl. Acids Res.* 38, W545-549.
- [Hsu2008] H. Hsu, J. A. Jones, and A. Orso. RAPID: Identifying bug signatures to support debugging activities. In ASE (Automated Software Engineering), 2008.
- [Huan2003] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraph in the presence of isomorphism, *Proceedings of the 3rd IEEE International Conference on Data Mining (ICDM)*, pp. 549-552, 2003.
- [Huan2004] J. Huan, W. Wang, D. Bandyopadhyay, J. Snoeyink, J. Prins, and A. Tropsha. Mining spatial motifs from protein structure graphs, *RECOMB*, pp. 308-315, 2004.
- [Huan2006] Huan J, Bandyopadhyay D, Prins J, Snoeyink J, Tropsha A, Wang W. Distance-based identification of spatial motifs in proteins using constrained frequent subgraph mining. *Proceedings of the LSS Computational Systems Bioinformatics Conference (CSB)*, pp. 227-238, 2006.
- [Jin2009] N. Jin, C. Young and W. Wang, Graph Classification Based on Pattern Co-occurrence, in *Proceedings of the ACM 18th Conference on Information and Knowledge Management (CIKM)*, pages 573-582, 2009.
- [Jin2010] N. Jin, C. Young and W. Wang, GAIA: graph classification using evolutionary computation, in *Proceedings of the ACM SIGMOD International Conference on management of Data*, pages 879-890, 2010.
- [Jin2011] Ning Jin, Wei Wang: LTS: Discriminative subgraph mining by learning from search history. *ICDE 2011*: 207-218.
- [Khan2010] A. Khan, X. Yan and K.-L. Wu, Towards Proximity Pattern Mining in Large Graphs, *SIGMOD'10* (Proc. 2010 Int. Conf. on Management of Data), June 2010.
- [Laskowski2005] Laskowski R.A., Watson J.D., and Thornton J.M. Protein function prediction using local 3d templates. *Journal of Molecular Biology*, 351:614–626, 2005.
- [Murzin1995] Murzin A. G., Brenner S. E., Hubbard T., Chothia C. (1995). SCOP: a structural classification of proteins database for the investigation of sequences and structures. *J. Mol. Biol.* 247, 536-540.
- [Ranu2009] S. Ranu and A. K. Singh. GraphSig: A Scalable Approach to Mining Significant Subgraphs in Large Graph Databases, in *Proceedings of the 25th International Conference on Data Engineering (ICDE)*, pages 844-855, 2009.
- [Saigo2008] Saigo H, Kraemer N, Tsuda K. Partial Least Squares Regression for Graph Mining, *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD2008)*, 578-586, 2008.

[Smalter2008] Smalter A, Huan J, Lushington G. A Graph Pattern Diffusion Kernel for Chemical Compound Classification. in Proceedings of the 8th IEEE International Conference on Bioinformatics and BioEngineering (BIBE'08), 2008.

[Smalter2009] Smalter A, Huan J, Lushington G, Graph Wavelet Alignment Kernels for Drug Virtual Screening, *Journal of Bioinformatics and Computational Biology*, Vol. 7 (3), pp. 473-497, 2009.

[Thoma2009] Thoma M, Cheng H, Gretton A, Han J, Kriegel H, Smola A, Song L, Yu P, Yan X, Borgwardt K. Near-optimal supervised feature selection among frequent subgraphs, *In SDM 2009*, Sparks, Nevada, USA.

[Yan2002] X. Yan and J. Han. gSpan: graph-based substructure pattern mining. In Proceedings of the 2002 IEEE International Conference on Data Mining, pages 721–724, 2002.

[Yan2003] Yan X and Han J. CloseGraph: mining closed frequent graph patterns. Proceedings of the 2003 ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD2003), 286-295, 2003.

[Yan2008] X. Yan, H. Cheng, J. Han, and P. S. Yu. Mining significant graph patterns by leap search. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 433–444, 2008.

[Yao2003] Yao H. et. al. An accurate, sensitive, and scalable method to identify functional sites in protein structures. *J. Mol. Biol.*, 326:255–261, 2003.

[Zhang2007] Zhang X, Wang W, Huan J. On demand Phenotype Ranking through Subspace Clustering Proceedings of SIAM International Conference on Data Mining (SDM), 2007.

[Zhang2008] Zhang S and Yang J. RAM: Randomized Approximate Graph Mining. Proceedings of the 20th international conference on Scientific and Statistical Database Management, pages 187-203, 2008.